High-Performance Generalized Tensor Operations: A Compiler-Oriented Approach

ROMAN GAREEV, Ural Federal University
TOBIAS GROSSER, ETH Zurich
MICHAEL KRUSE, INRIA, École Normale Supérieur, and Polly Labs

The efficiency of tensor contraction is of great importance. Compilers cannot optimize it well enough to come close to the performance of expert-tuned implementations. All existing approaches that provide competitive performance require optimized external code. We introduce a compiler optimization that reaches the performance of optimized BLAS libraries without the need for an external implementation or automatic tuning. Our approach provides competitive performance across hardware architectures and can be generalized to deliver the same benefits for algebraic path problems. By making fast linear algebra kernels available to everyone, we expect productivity increases when optimized libraries are not available.

CCS Concepts: • **Software and its engineering → Compilers**; • **Computing methodologies** → *Linear algebra algorithms*;

Additional Key Words and Phrases: Tensor contractions, high-performance computing, matrix-matrix multiplication

ACM Reference format:
Roman Gareev, Tobias Grosser, and Michael Kruse. 2018. High-Performance Generalized Tensor Operations: A Compiler-Oriented Approach. *ACM Trans. Archit. Code Optim.* 15, 3, Article 34 (August 2018), 27 pages.
https://doi.org/10.1145/3235029

1 INTRODUCTION

Tensor contraction (TC) is an important and widely used computational pattern. A general d-dimensional tensor $\mathcal{T} \in \mathbb{R}^{n_{u_0} \times \cdots \times n_{u_{d-1}}}$ can be defined as the set of scalar elements indexed by the set of indices $n_{u_0} \dots n_{u_{d-1}}$,

$$\mathcal{T} \equiv \{A_{n_{u_0} \cdots n_{u_{d-1}}} \in \mathbb{R} | (n_{u_0}, \dots, n_{u_{d-1}}) \in n_{u_0} \times \cdots \times n_{u_{d-1}}\}.$$

Let $\mathcal{A}$, $\mathcal{B}$, and C be d_A-, d_B-, and d_C-dimensional tensors, respectively. Let the free and the contracted indices of the tensor $\mathcal{A}$ be grouped into two bundles $I = i_0 \dots i_{r-1}$ and $P = p_0 \dots p_{t-1}$, respectively. Similarly, the indices of $\mathcal{B}$ are grouped into bundles $J = j_0 \dots j_{s-1}$ and P and the indices of C are grouped into bundles I and J. TC of tensors $\mathcal{A}$ and $\mathcal{B}$ into tensor C can be represented as $C_{\pi_C(IJ)} = \sum_P \alpha \cdot \mathcal{A}_{\pi_A(IP)} \cdot \mathcal{B}_{\pi_B(PJ)} + \beta \cdot C_{\pi_C(IJ)}$, where $\sum_P = \sum_{p_0=0}^{n_{p_0}-1} \dots \sum_{p_{t-1}=0}^{n_{p_{t-1}}-1}$, $\alpha, \beta \in \mathbb{R}$, n_{p_i} is the length of the tensor dimension that corresponds to the index p_i, and $\pi_C(IJ)$, $\pi_A(IP)$, and

Authors' addresses: R. Gareev, Ural Federal University, 19 Mira street, 620002 Ekaterinburg, Russia; email: gareev.roman@urfu.ru; T. Grosser, ETH Zürich, Department of Computer Science, Universitätstrasse 6, 8092 Zürich, Switzerland; email: tobias.grosser@inf.ethz.ch; M. Kruse, ENS - DI, 45 rue d'Ulm, 75005 Paris, France; email: michael.kruse@inria.fr.

1544-3566/2018/08-ART34 $15.00
https://doi.org/10.1145/3235029

$\pi_B(PJ)$ are permutations of the enclosed indices (Matthews 2016; Springer and Bientinesi 2018). TC is widely used in many scientific disciplines like machine learning (Abadi et al. 2015; Vasilache et al. 2014), spectral element methods (Tufo and Fischer 1999), and quantum chemistry calculations (Bartlett and Musiał 2007; Harrison et al. 2016). TC can be used to compute multidimensional Fourier transforms. For instance, a three-dimensional discrete Fourier transform (DFT) can be computed as the TC of two-dimensional DFTs (Pekurovsky 2012). In particular, the applications of general matrix-matrix multiplication TC of two-dimensional tensors include machine learning (Cong and Xiao 2014), data mining (Jayachandran and Venkatachalam 2016), quantum chemistry (Watson et al. 2010), and high-performance computing (Goto and Geijn 2008). The optimization of TC can be generalized to applications in the case of the matrix multiply-and-add operation (MMA[$\otimes$, $\oplus$]), which can be defined as $C \leftarrow \alpha \otimes C \oplus \beta \otimes A \otimes B$, where A, B, and C are three appropriately sized matrices, the $\oplus$ and $\otimes$ operations originate from the corresponding matrix semiring, and α and β are constants (Sedukhin and Paprzycki 2012). This allows solving algebraic path problems (APPs), such as finding the least and the most reliable paths, finding paths with maximum cost, or finding for all pairs the shortest connecting path. Efficiently computing TC is important in many situations.

Despite many years of development, none of the current leading production compilers and compiler front-ends (GCC (Stallman 1999), Clang (Lattner 2002), ICC (Intel 2015), IBM XL (IBM 2012)) can automatically transform a textbook-style implementation of TC into code that comes close to matching the performance of expert-tuned implementations. Common examples of matrix-matrix multiplication (MMA[$\times$, $+$]) that can be used to optimize TC (Matthews 2016; Springer and Bientinesi 2018) are available in the Basic Linear Algebra Subprograms (BLAS) (Dongarra et al. 1990; Lawson et al. 1979). A variety of specialized libraries (e.g., Intel's MKL (Intel [n.d.]), ARMPL (ARM 2015), BLIS (Van Zee and van de Geijn 2015), and OpenBLAS (Xianyi et al. 2012)) provide high-performance implementations for commonly used data types and hardware platforms. The existing approaches require previously optimized external code (e.g., routines of BLAS-compatible libraries) for deployment in advance but they can be used to replace code manually or automatically (Intel 2015; Menon and Pingali 1999) only if an optimized implementation is available for a given type and architecture. In situations where no pre-optimized code is available, the use of a textbook style implementation compiled with a state-of-the-art compiler usually achieves only a fraction of the theoretical machine performance.

In this study, we present a new compiler optimization for TC to narrow the difference in performance between compilers and approaches based on expert-tuned libraries (e.g., TBLIS (Matthews 2016), TCCG (Springer and Bientinesi 2018)). As part of a general purpose compilation flow, we identify a computational pattern to express TC and to automatically transform the original loop structure into a high-performance implementation. To obtain competitive performance, we apply multiple levels of tiling, as well as employing data-layout transformations to cache important sub-matrices in transposed form. By using an analytical performance model, we derive parameter values that exploit the computational resources of the machine employed for compiling, but without the need for iterative compilation or automatic tuning techniques. Our optimization was implemented in the Polly loop optimizer (Grosser et al. 2011) and it is available for any LLVM (Lattner 2002)-based static compiler, such that programs written in Fortran, C/C++, Julia, and various other languages immediately benefit.

Our contributions are:

- An automatic transformation for optimizing TC based on the computational structure proposed by the BLIS framework—including the necessary data-layout transformations (Section 4).
- A novel general infrastructure for performing arbitrary affine-linear data-layout transformations on a low-level compiler IR (Section 4.4).

- An analytical model is necessary for identifying the parameter values in MMA to compete with expert-tuned implementations (Section 4.5).
- Comparison of our approach to existing production compilers, vendor optimized BLAS libraries (Section 5) and approaches based on them. We attain the performance of the single-threaded instances of MMA[×, +] that are available in vendor optimized BLAS libraries and can reach more than 1.63× speedup over the Intel C compiler. In the case of APPs, we achieve more than 85% of the theoretical peak performance and 1.56× speedup over the code produced by the compilation. In the case of TC, we attain the performance of TCCG and TBLIS, 80.35% of the theoretical peak performance and a speedup of 84× compared with the Intel C loop optimizer.

2 BACKGROUND

In this section, we describe the polyhedral model, which is the basis of the algorithms presented in the following sections. First, we describe the mathematical foundations of the polyhedral model, before discussing its requirements and components. Listing 1 shows an example of a program containing a matrix-matrix multiplication of the form $C \mathrel{+}= A \times B$, where the sizes of matrices A, B, and C are $M \times K$, $K \times N$, and $M \times N$, respectively. We use this as an example to introduce the foundations of polyhedral loop modeling.

```
      for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
          for (p = 0; p < K; p++)
S:          C[i][j] += A[i][p] * B[p][j];
```

Listing 1. Example of matrix-matrix multiplication.

2.1 Mathematical Foundations of the Polyhedral Model

The polyhedral model is a mathematical framework for loop nest optimization, which focuses on modeling and optimizing the memory access behavior of a program and abstracts from individual computational operations (Feautrier and Lengauer 2011). $\mathbb{Z}$-Polytopes (Loechner and Wilde 1997), the integer points in a rational polyhedron, and Presburger relations, which are formulae defined recursively as the result of specific operations (e.g., boolean operations, quantified expressions, and comparisons) performed on constants and variables (Pugh and Wonnacott 1994a, 1994b), are the mathematical foundations of the polyhedral model. They help to define and operate on sets of the form $S = \{\vec{s} \in Z^d \mid f(\vec{s}, \vec{p})\}$, where $\vec{s}$ represents the integer tuples contained in the $\mathbb{Z}$-polytope, d is the dimensionality of the set, $\vec{p} \in Z^e$ is a vector of e parameters, and $f(\vec{s}, \vec{p})$ is a Presburger formula that evaluates to true *iff* $\vec{s}$ is an element of S for a given set of parameters $\vec{p}$. If a Presburger formula always evaluates to true, then the set S is called the universal set and it is abbreviated as $\{\vec{s}\}$.

2.2 SCoPs

A program region that satisfies the requirements of the polyhedral model is called Static Control Part (SCoP) (Bondhugula et al. 2008; Girbal et al. 2006). A SCoP is a set of program statements where the only control flow structures that it contains are if-conditions and one-dimensional for-loops with constant strides. It is assumed that each of these strides is equal to one; otherwise, the corresponding induction variable and loop bounds are modified. All of the loop bounds and conditionals of a SCoP are affine functions and affine inequalities, respectively, of the surrounding

loop iterators and global parameters. The global parameters are invariant during the execution of a SCoP, but their values are unknown at the compile time. Listing 1 is an example of a SCoP that contains the statement S.

SCoPs are defined in intermediate representations (e.g., LLVM IR (Lattner 2002), SSA (Stallman 1999)) used internally by compilers and compiler front-ends (e.g., Clang (Lattner 2002), GCC (Stallman 1999)), such that they can be extracted, optimized, and transformed into the corresponding intermediate representations (Grosser et al. 2012). To perform optimizations in production compilers where the compile time is a crucial factor, every basic block, which is a sequence of code with no branches in (out) except for the entry (at the exit), is represented as a SCoP statement. Although it prevents the independent scheduling of statements belonging to the same basic block, it provides greater scalability.

2.3 Components of the Polyhedral Representation

For SCoPs, program regions that satisfy the requirements of the polyhedral model (Section 2.2), each compute statement is described by three components: iteration domain, scheduling function, and access relation. These components are defined using $\mathbb{Z}$-polytopes (Loechner and Wilde 1997) and Presburger relations (Section 2.1).

An iteration domain describes the dynamic instances of the SCoP statement. It is represented as a $\mathbb{Z}$-Polytope defined by affine constraints on the global parameters and iteration variables of loops enclosing the SCoP statement. The iteration domain of the SCoP statement S from Listing 1 has the form $\{S(i,j,p) \mid 0 \leq i \leq M \wedge 0 \leq j \leq N \wedge 0 \leq p \leq K\}$.

A scheduling function defines the execution order of the individual dynamic instance of a SCoP statement. It is described with a $\mathbb{Z}$-polytope that relates dynamic statement instances to their execution time vectors. The lexicographical ordering of the execution time vectors defines the execution order for all dynamic instances. The scheduling function for the SCoP statement S from Listing 1 has the form $\{S(i,j,p) \rightarrow (i,j,p)\}$.

An access relation maps dynamic instances of a SCoP statement to the array element(s) that they access. It is described using a $\mathbb{Z}$-polytope that defines the relation between the iteration vector and the accessed array subscript. The pattern in which the array elements are accessed is evaluated using the stride of the access relation calculated with respect to the innermost loop. The stride is the distance between the memory accesses of two subsequently executed statement instances (Grosser et al. 2012). Accesses to scalar variables are modeled as accesses to zero-dimensional arrays. To distinguish the commonly multiple memory accesses of a SCoP statement, the access relations of the individual memory accesses are written in the expression evaluation order in the original program. The access relations describing the memory accesses of SCoP statement S from Listing 1 are $\{S(i,j,p) \rightarrow A(i,p)\}, \{S(i,j,p) \rightarrow B(p,j)\}, \{S(i,j,p) \rightarrow C(i,j)\}, \{S(i,j,p) \rightarrow C(i,j)\}$, where the first three relations represent reading from the memory and the last represents writing to the memory.

3 TC-LIKE KERNEL

We introduce a TC-like kernel as a set of programs with a data usage pattern that is similar to that produced by TC (Section 4). We show how it can be detected and mapped to matrices that may be optimized using approaches developed for MMA[×, +]. The TC-like kernel is defined as follows:

Definition 3.1. A TC-like kernel is a perfectly nested set of loops such that:

- It satisfies the requirements of the polyhedral model.
- Without loss of generality, it contains three nonempty bundles of one-dimensional for-loops with induction variables that are grouped into bundles $I = i_0 \ldots i_{r-1}$, $J = j_0 \ldots j_{s-1}$, and $P = p_0 \ldots p_{t-1}$, and they are incremented by one.

- The innermost loop body can be represented as a statement of the form $C_{\pi_C(IJ)} = E(A_{\pi_A(IP)}, B_{\pi_B(PJ)}, C_{\pi_C(IJ)})$, where $A_{\pi_A(IP)}$, $B_{\pi_B(PJ)}$, $C_{\pi_C(IJ)}$ are accesses to tensors $\mathcal{A}$, $\mathcal{B}$, C, respectively, $\pi_C(IJ)$, $\pi_A(IP)$, and $\pi_B(PJ)$ are permutations of the enclosed indices, and E is an expression that contains reads from the tensors $\mathcal{A}$, $\mathcal{B}$, C, and an arbitrary number of reads from constants with respect to bundles I, J, and P.

According to the definition of a TC-like kernel, the following sufficient conditions for a SCoP to be a TC-like kernel can be stated. It has only one SCoP statement S as well as t true dependencies, t anti-dependencies, and t output dependencies. The dependencies have the form $\forall i \in 0 \dots t-1$ $S((p_0, \dots, p_i, n_{p_{i+1}} - 1, \dots, n_{p_{t-1}} - 1) * \pi(IJ)) \rightarrow S((p_0, \dots, p_i + 1, 0, \dots) * \pi(IJ))$, where $t_1 * t_2$ is a unique tuple such that each entry of tuples t_1 and t_2 appears only once and they appear in the same order as in the underlying set. Thus, it is legal to interchange the loops of I and J. In the case of P, it is necessary to check that either bundle P contains only one element or an associative operation is used to update C; otherwise, interchanging the loops of bundle P can violate the dependencies. According to the form of the innermost loop body, the SCoP should contain reads of the form $S(\dots) \rightarrow \mathrm{A}_{\pi_A(IP)}$, $S(\dots) \rightarrow \mathrm{B}_{\pi_B(PJ)}$, $S(\dots) \rightarrow \mathrm{C}_{\pi_C(IJ)}$, and possibly an arbitrary number of reads that have stride zero with respect to the loops with the induction variables of I, J, and P. The access relations of the individual memory accesses are written in the expression evaluation order of the original program (Section 2), so the last access relation should have the form $S(\dots) \rightarrow \mathrm{C}_{\pi_C(IJ)}$ and correspond to the last memory access in S that writes to memory.

To detect TC-like kernels, it is sufficient to check the specified conditions. We can verify that the last memory access is a store that writes the result from the TC-like kernel. It allows us determine the union of bundles I and J. Subsequently, we can check that $\forall i \in 0 \dots t-1$, we only have true dependencies and anti-dependencies of the form $S((p_0, \dots, p_i, n_{p_{i+1}} - 1, \dots, n_{p_{t-1}} - 1) * \pi(IJ)) \rightarrow S((p_0, \dots, p_i + 1, 0, \dots) * \pi(IJ))$. This can also help to determine bundle P. Next, we can check that the SCoP contains at least three read accesses and determine bundles I and J. We can verify that all additional read memory accesses that do not correspond to the operands of the TC-like kernel have stride 0 if the innermost loop is exchanged with any of the loops. Finally, we can check that either bundle P contains only one element, or we use an associative operation in the case of generalized summation.

To optimize the detected TC-like kernels, we employ an approach that allows the mapping of tensors to matrices and apply optimized MMA[×, +] routines (Matthews 2016). This method is based on the new data layout, which provides a mapping between the elements of tensors and their locations in memory. An example of a commonly used data layout for matrices is the row-major data layout, where the elements are assigned successive locations moving across the first row and then continuing across subsequent rows. In the case of tensors, this can be generalized such that the tensor element $T_{i_0 \dots s i_{d-1}}$ for T of the shape $n_{i_0} \times \dots \times n_{i_{d-1}}$ is at the location $\sum_{k=0}^{d-1} i_k \prod_{l=k+1}^{d-1} n_{i_l}$. For example, if its dimensions are grouped into two bundles I and J, then the location can be represented as $\sum_{k=0}^{d_I-1} i_k (\prod_{l=k+1}^{d_I-1} n_{i_l})(\prod_{l=0}^{d_J-1} n_{j_l}) + \sum_{k=0}^{d_J-1} j_k \prod_{l=k+1}^{d_J-1} n_{j_l} = (\sum_{k=0}^{d_I-1} i_k \prod_{l=k+1}^{d_I-1} n_{i_l}) n_J + \sum_{k=0}^{d_J-1} j_k \prod_{l=k+1}^{d_J-1} n_{j_l} = \overline{I} n_J + \overline{J}$. If we consider the location of element M_{ij} for an M×N matrix, $i \cdot n_j + j$, then we can deduce formulae for computing $\overline{I}$, n_j, and $\overline{J}$ for the new data layout, which are $\sum_{k=0}^{d_I-1} i_k \prod_{l=k+1}^{d_I-1} n_{i_l}$, $\prod_{l=0}^{d_J-1} n_{j_l}$, and $\sum_{k=0}^{d_J-1} j_k \prod_{l=k+1}^{d_J-1} n_{j_l}$, respectively. Subsequently, new induction variables can be used to perform any type of logical matrix operation (Matthews 2016).[1]

[1]The polyhedral representation allows the simplification of the original approach, which requires the storage of additional vectors of length $\mathrm{O}(n_I + n_J + n_P)$ for storing the information about strides for the new data layout (Matthews 2016). In particular, the access relations contain the information about the accessed arrays (e.g., sizes of dimensions), so this representation can be reused to avoid additional vector allocation.

In our case, the same formulae can be used to modify the scheduling function for the SCoP statement representing the TC, thereby operating on new induction variables and matrices instead of tensors. In the following sections, for the sake of simplicity, we let the innermost loop body of the TC-like kernel have the form C[i][j] = E(A[i][p], B[p][j], C[i][j]), where i, j, and p are induction variables, A[i][p], B[p][j], and C[i][j] are accesses to matrices A, B, and C, respectively, and E is an expression that contains reads from the matrices. This representation allows us to consider only the algorithm for the optimization of MMA[×, +] and shows that it can be applied to TC-like kernels (Section 4).

4 OPTIMIZING A TC-LIKE KERNEL

In this section, we present an algorithm for transforming the TC-like kernel into an implementation that is structured in a similar manner to an expert-optimized kernel. First, we describe the expert-designed MMA[×, +] implementation in BLIS (Van Zee and van de Geijn 2015) and then discuss our optimization, which uses general purpose compiler transformations based on polyhedral modeling to obtain a MMA[×, +] implementation with comparable performance to an expert-optimized kernel.

```
Loop1: for (j = 0; j < N; j += Nc)
Loop2:   for (p = 0; p < K; p += Kc) {
           // Pack into Bc
           B(p:p + Kc - 1, j:j + Nc - 1) → Bc
Loop3:     for (i = 0; i < M; i =+ Mc) {
             // Pack into Ac
             A(i:i + Mc - 1, p:p + Kc - 1) → Ac
             // Macro-kernel
Loop4:       for (jc = 0; jc < Nc; jc += Nr)
Loop5:         for (ic = 0; ic < Mc; ic += Mr)
                 // Micro-kernel
Loop6:           for (pc = 0; pc < Kc; pc++)
                   Cc(ic:ic + Mr - 1, jc:jc + Nr - 1)
                     += Ac(ic:ic + Mr - 1, pc) · Bc(pc, jc:jc + Nr - 1)
           }
         }
```

Listing. 2. The implementation of MMA[×, +] of BLIS.

4.1 Expert Implementation of MMA[×, +]

An expert implementation of the MMA[×, +] algorithm is an implementation tuned by dense linear algebra experts. Examples of these implementations can be found in GotoBLAS (Goto and Geijn 2008) and its successors, OpenBLAS (Xianyi et al. 2012) and BLIS (Van Zee and van de Geijn 2015). In contrast to GotoBLAS and OpenBLAS, BLIS exposes three innermost loops that facilitate the analytical identification of optimal parameters (Van Zee and van de Geijn 2015).

The implementation of MMA[×, +] in BLIS comprises two packing routines and five loops around the micro-kernel. The micro-kernel is a loop around an outer product, which can be implemented in assembly. The micro-kernel and two surrounding loops form the macro-kernel. The pseudo-code for the implementation[2] can be found in Listing 2. The determination of the M_c, N_c, K_c, M_r, and N_r parameters is considered in Section 4.5.

[2]We use the mathematical notations for matrices A_c, B_c, C_c presented by (Low et al. 2016).

4.2 Optimization of a TC-Like Kernel

In this subsection, we present a new polyhedral optimization that helps to obtain code, which is a generalization of the implementation described by Section 4.1 in terms of the outer product defined in the semiring representing MMA[×, +]. The optimization transforms the SCoP statement containing the TC-like kernel (Section 3) by creating the micro- and macro-kernels, and by performing the packing transformation described in Section 4.3. The overall algorithm is summarized in Algorithm 1.

ALGORITHM 1: Algorithm for Optimizing a TC-Like Kernel.

Input: SCoP statement S that represents the TC-like kernel and, consequently, can be represented as C[i][j] = E(A[i][p], B[p][j], C[i][j]), where i, j, p are induction variables, A[i][p], B[p][j], C[i][j] are accesses to matrices *A*, *B*, *C*, respectively, and E is an expression that contains the reads from the matrices.

1 Identify individual loops with induction variable names.
2 Interchange the loops in the loop nest such that i, j, and p are the innermost loops in the following order: j, p, and i, where i has the highest level in the loop nest.
3 Tile i, j, and p with M_c, N_c, and K_c, respectively, to produce i_c, j_c, and p_c, and interchange i_c and p_c.
4 Tile i_c, j_c, and p_c with M_r, N_r, and 1, respectively, to produce i_r, j_r, and p_r, and eliminate p_r.
5 Separate the domains of i_r and j_r into full and partial tiles, and unroll j_r and i_r.
6 Perform the packing transformation.
7 Vectorize the code in p_c.
8 Hoist and sink read and write accesses to the matrix, which stores the result of the TC-like kernel, from p_c, respectively.

Output: optimized code

Let us consider the algorithm and compare the code it produces as well as the implementation of MMA[×, +] of BLIS described in Section 4.1. First, we obtain the three loops around the macro-kernel using interchanging and tiling transformations (Lines 1–3). We then produce the macro-kernel by tiling the innermost loops (Line 4). Subsequently, we separate the domains of the innermost two loops into full and partial tiles to allow the vectorization in the case of parametric bounds and loop sizes that cannot be divided evenly by the tile sizes (Line 5). Finally, we create the micro-kernel by applying unrolling and vectorization of the innermost loops, and performing the packing transformation (Lines 6–8). Consequently, the generated code is similar to the expert implementation (Section 4.1), except the innermost loop body that is not necessarily the outer product.[3]

Let us consider the innermost loop produced by the transformation. According to Section 3, this loop updates the blocks of the matrix, e.g., *C*, with length M_r and width N_r. Updating is achieved by a series of K_c calculations involving M_r elements from the packed matrix, e.g., A_c, and N_r elements from the packed matrix, e.g., B_c. At each iteration of p_c, $\mathrm{M}_r\mathrm{N}_r$ intermediate results are computed by operating on M_r elements from the matrix A_c and N_r elements from the matrix B_c. Subsequently, they are accumulated into the block of the matrix *C*, e.g., C_c, which is reused between iterations in loop p_c. There is no reuse of the elements of A_c and B_c in p_c, because they are used exactly

[3]In the case of MMA[×, +], the implementation obtained is structured in a similar manner to the implementation of MMA[×, +] in BLIS. However, the micro- and macro-kernels generated by the optimization cannot be used instead of the corresponding parts of the BLIS framework. To reuse the micro-kernel, it should be generated as the separate function required by BLIS.

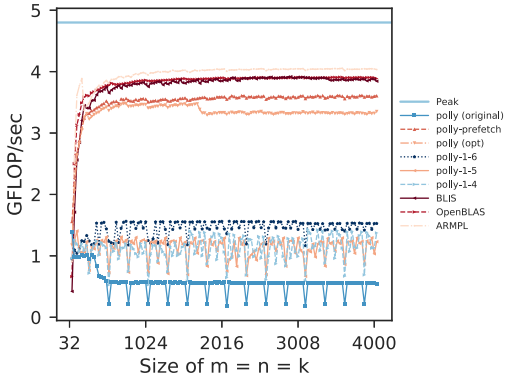

Fig. 1. MMA[×, +] for ARM.

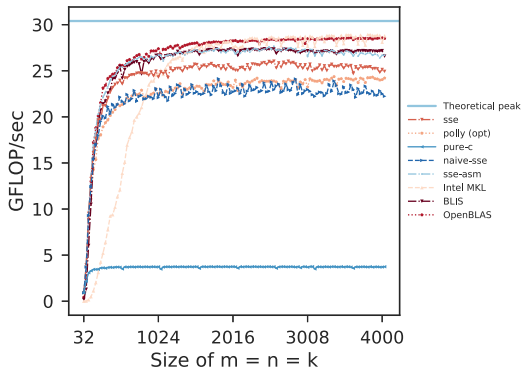

Fig. 2. MMA[×, +] for Sandy Bridge.

once each time via p_c. Performing this analysis on the remaining loops can show that our data usage pattern is similar to that produced by the expert implementation described in Section 4.1. Consequently, since the size of reused data becomes smaller as the loop depth increases, smaller amounts of reused data are mapped to faster memory layers in the memory hierarchy (Goto and Geijn 2008).

As an example, we applied the algorithm to the statement for the SCoP presented in Listing 1. In addition, Figure 1 presents the single-threaded performance evaluation results for the steps in the implementation of MMA[×, +] provided by the PolyBench 3.2 benchmark suite (Pouchet 2011), matrices containing elements of type double, and ARM (see Section 5 for details).

Applying the interchanging and tiling transformations (Lines 1–3) transforms the output dimensions of the scheduling function $\{S(i,j,p) \rightarrow (i,j,p)\}$ to $\lfloor \frac{j}{N_c} \rfloor, \lfloor \frac{p}{K_c} \rfloor, \lfloor \frac{i}{M_c} \rfloor, j \bmod N_c, i \bmod M_c, p \bmod K_c$.[4] After tiling the innermost loops (Line 4) $j \bmod N_c, i \bmod M_c, p \bmod K_c$ are transformed into $\lfloor \frac{j \bmod N_c}{N_r} \rfloor \lfloor \frac{i \bmod M_c}{M_r} \rfloor$, $p \bmod K_c$, and the dimensions $j \bmod N_r, i \bmod M_r$ are added. The transformed SCoP can be found in Listing 3. For simplicity, $M \bmod M_c = N \bmod N_c = K \bmod K_c = 0$. The loop tiling and loop interchange transformations (Lines 1–4) help to achieve 23% of the theoretical peak (polly-1-4).

Loop unrolling transformations (Line 5) do not affect the performance (polly-1-5). The packing transformation (Line 6) helps to achieve 26% of the theoretical peak (polly-1-6). The transformed SCoP can be found in Listing 4. The result obtained by the unrolling and packing transformation (Lines 5 and 6) is described in Section 4.3.

In the case of Polly, the implementation of vectorization and the subsequent loop hoisting and loop sinking (Lines 7 and 8) are based on the LLVM's vectorization passes, and they allow us to achieve 73% of the theoretical peak (polly (opt)). To improve the performance further, prefetch instructions can be generated using the loop data prefetch pass in LLVM, which is not used by default and it is not available for all targets (e.g., x86_64 architecture) at the time of writing this article. In the case of ARM, it helps to achieve 75% of the theoretical peak. We conclude that the performance of the micro-kernel is degraded by the lack of prefetching instructions. Nevertheless, Figure 1 shows that using prefetch instructions is not sufficient to match the performance of optimized libraries.

[4]If there is only one constraint on a newly introduced dimension that has the form of an equality and it does not include other newly introduced dimensions, then we replace the dimension with the expression containing only the original dimensions for the sake of simplicity.

```
for (j = 0; j <= N/Nc; j += 1)
  for (p = 0; p <= K/Kc; p += 1)
    for (i = 0; i <= M/Mc; i += 1)
      for (jc = 0; jc <= Nc/Nr; jc += 1)
        for (ic = 0; ic <= Mc/Mr; ic += 1)
          for (pc = 0; pc <= Kc; pc += 1)
            for (jr = 0; jr <= Nr; jr += 1)
              for (ir = 0; ir <= Mr; ir += 1)
                S(Mc * i + Mr * ic + ir,
                  Nc * j + Nr * jc + jr,
                  Kc * p + pc);
```

Listing 3. Optimized MMA[×, +].

```
for (j = 0; j <= 31; j += 1)
  for (p = 0; p <= 31; p += 1) {
    for (tmp1 = 32 * j; tmp1 <= 32 * j + 31; tmp1 += 1)
      for (tmp2 = 32 * p; tmp2 <= 32 * p + 31; tmp2 += 1)
        Copy0(0, tmp1, tmp2);
    for (i = 0; i <= 31; i += 1) {
      for (tmp1 = 32 * i; tmp1 <= 32 * i + 31; tmp1 += 1)
        for (tmp3 = 32 * p; tmp3 <= 32 * p + 31; tmp3 += 1)
          Copy1(tmp1, 0, tmp3);
      for (jc = 0; jc <= 15; jc += 1)
        for (ic = 0; ic <= 15; ic += 1)
          for (pc = 0; pc <= 31; pc += 1) {
            S(32 * i + 2 * ic, 32 * j + 2 * jc, 32 * p + pc);
            S(32 * i + 2 * ic + 1, 32 * j + 2 * jc, 32 * p + pc);
            S(32 * i + 2 * ic, 32 * j + 2 * jc + 1, 32 * p + pc);
            S(32 * i + 2 * ic + 1, 32 * j + 2 * jc + 1, 32 * p + pc);
          }
    }
  }
```

Listing 4. Optimized MMA[×, +].

According to previous studies that analyzed high-performance implementations of MMA[×, +] (Lehn 2014), effective register allocation and instruction scheduling may be necessary. We followed this guidance to manually optimize MMA[×, +] and determine what prevents us from always achieving high performance. Figure 2 presents the results of single-threaded performance evaluations of MMA[×, +], matrices containing elements of type double, and Sandy Bridge (see Section 5 for details). The results present the best performance obtained for GCC and Clang.

pure-c represents the evaluation of the implementation presented in Listing 1 with loop interchange transformations (Lines 1–4) as well as the application of the packing transformation (Line 6), which helps to achieve 12.31% of the theoretical peak.

Loop unrolling transformations (Line 5) and manual vectorization using SSE intrinsics allows us to achieve 79.74% of the theoretical peak (naive-sse). It generates a micro-kernel that can be represented as $C_c \leftarrow C_c + [\,A_c(I, p_c)\mathbb{B}_c\,]$, where $I \in i_c \ldots i_c + M_r - 1$, $\mathbb{B}_c$ is $B_c(p_c, j_c{:}j_c + N_r - 1)$. It should be noted that the elements of the second operand of the matrix-matrix multiplication can be stored in a vector register to increase the performance (Lehn 2014). At the time of writing this article, the LLVM vectorizer is not capable of performing these transformations. Thus, our optimization (Lines 1–8) generates the representation of the micro-kernel.[5]

To improve the performance further and achieve 83.91% of the theoretical peak (sse), we also keep the elements of the matrix A in a vector register and use different permutations of its elements to obtain all of the rank-1 updates performed by the micro-kernel (Section 4.1). This process generates a micro-kernel that can be represented as $C_c \leftarrow C_c + A_c(i_c : i_c + M_r - 1, p_c) \cdot B_c(p_c, j_c : j_c + N_r - 1)$.

To avoid the limitations of SSE intrinsics, we translate the intrinsics to assembly by ourselves. It allows us to manually perform instruction scheduling to consider the latency of the SSE instructions and fully occupy the execution units of the CPU. For instance, there are several loads in the vector registers in the innermost loop but instead of waiting for all of them to be completed, we can execute FMA operations between the load calls. Furthermore, manual translation of the SSE

[5]The values of M_r and N_r help to avoid the introduction of stalls in the pipeline (Low et al. 2016). They can be swapped without violating this property (Low et al. 2016). Since the LLVM's vectorization pass stores only the first operand of the matrix-matrix multiplication in a vector register, we swap the values of M_r and N_r to increase the number of elements stored to vector registers.

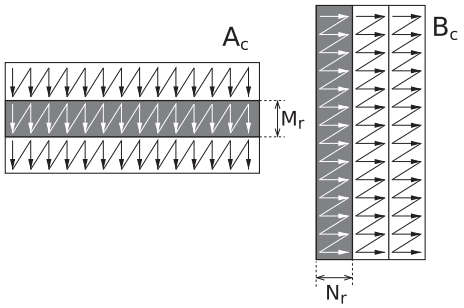

Fig. 3. Packing in the BLIS and GotoBLAS implementations of MMA[×, +].

intrinsics and additional unrolling of the innermost loop of the micro-kernel allows us to use as many of the vector registers as possible without modifying the data usage pattern. In addition, we use prefetch instructions to prefetch the elements of the operands of MMA from two iterations ahead, which helps to achieve 86.92% of the theoretical peak and match the performance of the code in the BLIS framework for all data sizes (sse-asm). The instruction scheduling and register allocation processes depend on the LLVM's vectorization passes as well as the compiler backends, which produce code for the specified machine or other languages. In addition to the previously described problem related to micro-kernel generation, a problem with register allocation occurs. Specifically, the LLVM's vectorization passes neither additionally unroll the innermost loop of the micro-kernel or vectorize this loop if it is unrolled by Polly. Improving these parts of the compiler are outside the scope of this study.

4.3 Packing Transformation

The copying of the data in matrices A and B, the operands of the TC-like kernel into the created arrays A_c and B_c, respectively, during the packing transformation (Van Zee and van de Geijn 2015) is illustrated in Figure 3. In our case, A_c and B_c are three-dimensional arrays with sizes of $\frac{M_c}{M_r} \times K_c \times M_r$ and $\frac{N_c}{N_r} \times K_c \times N_r$, respectively, which helps to ensure that their elements are read during in-stride access, aligned to the cache line boundaries, and preloaded into certain cache levels. The packing transformation can be represented as a data-layout transformation (Section 4.4), which introduces a new array, copies data to it, and changes the memory access locations to reference the array.

We now consider how the packing transformation can be expressed as a sequence of transformations with a polyhedral model to produce a SCoP (Section 2). The corresponding access relation should be created to introduce a new array. Data can be copied to an array by introducing the SCoP statement that contains a read from the specific location in memory and storing it to another. This transformation requires updating the scheduling functions for all the SCoP statements as well as the iteration domains and access relation maps if new loops are introduced. The corresponding access relations should be modified to change a memory access location. Since the described transformations can be performed with the polyhedral model without violations of its requirements, the result of the packing transformation is the new SCoP.

As an example, we apply the packing transformation to the statement of the SCoP presented in Listing 1. We assume that Lines 1–4 of Algorithm 1 are applied. The modified access relations help to map every element of matrices A and B to the corresponding elements of matrices Ac and Bc, respectively, and they represent relations of the form: $S(i, j, p) \rightarrow A_c(\lfloor \frac{i \bmod M_c}{M_r} \rfloor, p \bmod K_c, i \bmod$

$M_r), S(i,j,p) \rightarrow B_c(\lfloor \frac{j \bmod N_c}{N_r} \rfloor, p \bmod K_c, j \bmod N_r)$. The transformed SCoP can be found in Listing 4, where the SCoP statements Copy_0 and Copy_1 represent copying the elements to arrays B_c and A_c, respectively. For simplicity, M = N = K = 1,024, $M_c = N_c = K_c = 32$, and $N_r = M_r = 2$. Their access relations represent relations of the form: $\text{Copy}_0(i,j,p) \rightarrow B(p,j)$, $\text{Copy}_0(i,j,p) \rightarrow B_c(\lfloor \frac{j \bmod N_c}{N_r} \rfloor, \quad p \bmod K_c, j \bmod N_r)$, $\text{Copy}_1(i,j,p) \rightarrow A(i,p), \text{Copy}_1(i,j,p) \rightarrow A_c(\lfloor \frac{i \bmod M_c}{M_r} \rfloor, p \bmod K_c, i \bmod M_r)$.

4.4 Data-Layout Transformation Infrastructure

Data layout transformation comprises a class of transformations that optimizes the layout of the data (Srikant and Shankar 2007), which can be conducted within either a single aggregate data type or across data objects, e.g., to ensure that they are read during in-stride access, aligned to cache line boundaries, and preloaded into certain cache levels (Section 4.3). New data layout can eliminate the memory stream alignment issue, i.e., redundant loads where an element is moved with a different load for every distinct vector register position where it needs to be used (Henretty et al. 2011). Consequently, data layout transformations can help to decrease the memory access latency.

We extend Polly (Grosser et al. 2011) to perform polyhedral data-layout optimization. In particular, we add the ability to change, create access relations, and introduce SCoP statements, each of which represents copying elements from one array to another. We check that these transformations do not violate the requirements of a polyhedral model. Hence, the result obtained after their application is a new SCoP. Furthermore, we extend JSCoP, which is a file format based on JSoN (Grosser et al. 2012), to allow affine-linear data-layout transformations. Thus, we can test the transformations manually without any knowledge of the compiler's internals as well as developing new research prototypes.

To the best of our knowledge, we developed the first general infrastructure for polyhedral affine-linear data-layout transformations, which allows their application in a production compiler. We use it to implement the packing transformation, which requires the introduction of a new array, copies data to it, and changes the memory access locations to reference the array (Section 4.3). Even without additional modifications, this approach can be used to implement existing data-layout transformations, such as dimension-lifted transposition, which helps to eliminate the memory stream alignment issue (Henretty et al. 2011).

4.5 Architecture Model

To obtain a high-performance implementation of a TC-like kernel, particulary MMA[×, +], the optimization should be mapped onto the architecture. A model of the hypothetical processor can be created for this purpose. For example, the analytical model for determining the blocksize parameters in BLIS (Low et al. 2016) describes caches, vector instructions, load/store architecture, and vector registers, as follows.

— **Caches.** All data caches are set-associative. Each cache level Li is characterized by the size of the cache line C_{Li}, associativity degree W_{Li}, size S_{Li}, and the number of sets N_{Li}.
— **Vector instructions.** The throughput of the processor floating-point arithmetic units is an N_{VFMA} vector fused multiply-add instructions (VFMA) per clock cycle.[6] The minimum

[6]Information about latencies and throughputs can be found in manuals with instruction tables (ARM 2016; Fog 2017). In some cases, they contain the reciprocals of N_{VFMA}. For example, in the case of the Intel Xeon Phi 7210, N_{VFMA} is 2 and thus the reciprocal of the throughput is 0.5 (Fog 2017).

number of cycles between the execution of two dependent consecutive VFMA instructions is L_{VFMA}, which is the latency of each VFMA. In the case of architectures without VFMA instructions, L_{VFMA} is the sum of the latencies for one multiply instruction and one addition instruction.

— **Load/store architecture and vector registers.** Data are loaded into the processor registers before computations are performed based on them. Each vector register can hold N_{VEC} elements of size S_{DATA}.

Our optimization uses the modified version of the model, where instead of L_{VFMA}, we use L_{VMMA}, which is the sum of the latencies for instructions comprising vectorized matrix multiply-and-add operations (VMMA). Instead of N_{VFMA}, we consider N_{VMMA}, which is the processor throughput for the VMMA per clock cycle (Sedukhin and Paprzycki 2012). If the vector instructions comprising the VMMA can be executed simultaneously, then N_{VMMA} can be estimated as $N_{VMMA} = \min(N_1, N_2)$, where N_i is the throughput of an instruction comprising the VMMA; otherwise, it can be estimated as $N_{VMMA} = \frac{\min(N_1, N_2)}{2}$. This is explained by the ability to interchange independent instructions for different VMMA instructions that comprise a long sequence. These sequences can be used to estimate N_{VMMA} (Fog 2017).

For example, in the case of Intel Sandy Bridge, double floating point data type, MMA[×, *min*], MMA[×, *max*], and MMA[×, −], the corresponding instructions do not require the same execution unit so they can be executed simultaneously (Fog 2017). The throughputs of these instructions are equal to 1 and the sum of their latencies are equal to 8, so the values of N_{VMMA} and L_{VMMA} are 1 and 8, respectively. Following the same logic, in the case of MMA[/, *max*], the values of N_{VMMA} and L_{VMMA} are 0.05 and 48, respectively. In the cases of MMA[+, *max*], MMA[+, *min*], and MMA[*min*, *max*], the corresponding instructions require the same execution unit. The throughputs of these instructions are equal to 1 and the sum of their latencies are equal to 6, so the values of N_{VMMA} and L_{VMMA} are 0.5 and 6, respectively.

Our optimization produces code with a similar data usage pattern to that used in the implementation of MMA[×, +] in BLIS (Section 4.2). Hence, after determining the values of the parameters, we can use the same strategy that is applied to BLIS (Low et al. 2016) to deduce the formulae needed to find the values of the parameters for the micro- and macro-kernels. M_r and N_r are the parameters of the micro-kernel and they can be computed as follows: $M_r = \lceil \frac{\sqrt{N_{VEC} L_{VMMA} N_{VMMA}}}{N_{VEC}} \rceil N_{VEC}$ and $N_r = \lceil \frac{N_{VEC} L_{VMMA} N_{VMMA}}{M_r} \rceil$. They are set at sufficiently large values to avoid stalls in the floating-point pipelines caused by executing dependent consecutive VMMA during the updating of the matrix, e.g., C, by the micro-kernel (Section 4.2). In addition, M_r and N_r are set at the smallest values to release the vector registers for the elements of matrices A and B, which are the operands of the TC-like kernel.

The size of the reused data decreases as the loop depth increases, so smaller reused data are mapped onto faster memory layers in the memory hierarchy, where the faster layers have a lower capacity than the slow layers (Section 4.2). To allow rapid access and reduce cache trashing, the reused data should ideally be kept in the cache between iterations. Thus, there are restrictions on the values of the parameter of the micro-kernel K_c, and the parameters of the macro-kernel M_c and N_c, which define the shape of reused data (Section 4.1). We use the following formulae to compute these parameters: $K_c = \frac{C_{A_r} N_{L1} C_{L1}}{M_r S_{DATA}}$, where $C_{A_r} = \lfloor \frac{W_{L1}-1}{1+\frac{N_r}{M_r}} \rfloor$, $M_c = \frac{(W_{L2}-2) S_{L2}}{K_c S_{DATA} W_{L2}}$, and $N_c = \lfloor \frac{C_{B_c}}{K_c S_{DATA} N_r} \rfloor N_r$, where C_{B_c} is the number of bytes of memory available for array B_c created during the packing transformation (Section 4.3). These parameters are selected to conform to the sizes of the caches, the cache replacement policy, and the cache organization.

These formulae are similar to those applied in the case of BLIS (Low et al. 2016) and they are deduced in a similar manner. The exceptions are N_{VMMA} and L_{VMMA}, which are used instead of N_{VFMA} and L_{VFMA}, respectively. It allow us to apply the optimization to MMA[$\otimes$, $\oplus$] and to solve the APP, such as finding the least and most reliable paths, or finding the paths with the maximum cost. In the case of MMA[$\times$, +], the values derived by our analytical model can be optimal, because N_{VMMA} and L_{VMMA} are equal to N_{VFMA} and L_{VFMA}, respectively.

4.6 Scalability

Parallelism is important for high-performance modern processing platforms and it is essential in the case of big data problems (Facchinei et al. 2014). In this study, we restrict ourselves to the case of data level parallelism and a single-threaded implementation of a TC-like kernel. In this subsection, we briefly describe the issues and possibilities related to the scalability of the proposed approach.

According to the expert multithreaded implementation of MMA[$\times$, +] (Smith et al. 2014), j_c, the second loop around the micro-kernel (Listing 2), can provide a good opportunity for parallelization. In particular, if the ratio of N_c relative to the parameter N_r is large, which is usually the case, then the time spent in this loop cancels out the cost of packing the elements of the created array A_c (Section 4.3) into the L2 cache (Smith et al. 2014). Polly automatically checks all of the generated loops and introduces OpenMP parallelism for the outermost parallel loops (Grosser et al. 2011). In the case of a TC-like kernel, particularly MMA[$\times$, +], i, j, and p contain the packing transformation (Section 4.3), which introduces dependencies that prevent parallelization. Thus, j_c is the outermost parallel loop that is automatically parallelized by Polly. Section 5 presents the performance evaluation results for the proposed approach.

However, if the L2 cache is not shared and j_c is parallelized, then the elements of the created A_c are duplicated across the L2 caches. This occurs during the execution of the micro-kernel and it may overlap with the computation, which can reduce the performance (Smith et al. 2014). In this case, the first loop around the macro-kernel i, can be considered. If we parallelize this loop, then each thread will be assigned different elements of the matrix A, which reside in the L2 cache, and they are packed into the created array A_c, where this may cause a race condition. In the case of Polly, arrays A_c and B_c are allocated statically during compilation. Thus, they can be replicated to avoid the race condition by using OpenMP parallelism. However, Polly does not support this mechanism at present.

Determining the optimal parameter values for a multithreaded implementation of MMA[$\times$, +] as well as the loops that should be parallelized to obtain the best performance gain is orthogonal to the analytical model for determining the blocksize parameters in BLIS (Low et al. 2016). The modified version of this model (Section 4.5), which is used by the proposed approach, is affected by the same issue.

5 EXPERIMENTAL RESULTS

In this section, we present the results of the performance evaluations for the proposed optimization, TCCG (Springer and Bientinesi 2018), TBLIS (Matthews 2016), and the current production compilers, as well as the compiler front-end (GCC (Stallman 1999), Clang (Lattner 2002), ICC (Intel 2015), and IBM XL (IBM 2012)). In addition, we compared the performance of the code generated using our approach and the instances of MMA[$\times$, +] that are available in Basic Linear Algebra Subprograms (Intel's MKL (Intel [n.d.]), ARMPL (ARM 2015), BLIS (Van Zee and van de Geijn 2015), and OpenBLAS (Xianyi et al. 2012)). We also showed that the approach can be applied to APPs.

Our experimental setup (Tables 1 and 2) comprised modern processors (IBM Power 8, Intel Xeon Phi, Intel Sandy Bridge, Intel Kaby Lake, and APM883208-X1) with different architectures (ppc64le, x86_64, and aarch64). Each single measurement or result reported in this section is the

Table 1. Experimental Setup

Nickname	CPU	Clock rate (GHz)	RAM (GB)	N_{VEC} double	L_{VFMA}[7, 8]	N_{VFMA}[7]	S_{L_1} (Kbytes)	W_{L_1}	S_{L_2} (Kbytes)	W_{L_2}
Intel Sandy Bridge	Intel Core i7-3820	3.6(3.8)[9]	16	4	8	1	32	8	256	8
Intel Kaby Lake	Intel Core i7-7700	3.6(4.2)[9]	32	4	4(6)	2	32	8	256	8
Intel Xeon Phi	Intel Xeon Phi 7210	1.3	110	8	6(7)	2	32	8	1,024	16
IBM Power 8	POWER8NVL	4.023	256	2	5.5(12)	2	64	8	512	8
ARM	APM883208-X1	2.4	32	2	9(36)	0.5	32	8	256	8
Intel Xeon E5	Intel Xeon E5-2630 v4	2.2(3.1)[9]	64	4	5(6)	2	32	8	256	8

Table 2. The Software Version and Additional Options

Nickname	Version	Additional options
polly (original)	6.0.0	-O3 -march=native[10] -mllvm -polly
polly (opt)	6.0.0	-O3 -march=native -mllvm -polly[11] -polly -target-throughput-vector-fma=N_{VMMA} -mllvm -polly-target-latency-vector-fma=L_{VMMA} -mllvm -polly-target-1st-cache-level-associativity=W_{L_1} -mllvm -polly-target-2nd-cache-level-associativity=W_{L_2} -mllvm -polly-target-1st-cache-level-size=S_{L_1} -mllvm -polly-target-2nd-cache-level-size=S_{L_2} -ffp-contract=fast -ffast-math
clang	6.0.0	-O3 -march=native
gcc	4.9.2	-O3 -march=native
icc	17.0.2	-O3 -march=native
IBM XLC	13.1.5	-O3 -qarch=auto -qtune=auto
Intel MKL	11.3.3	
BLIS	0.2.2	
OpenBLAS	0.2.19	
ARMPL	2.4.0	
TCCG	0.1.2	
TBLIS	1.0.0	

corresponding arithmetic mean. We collected measurements until the 95% confidence intervals were within 10% of our reported means.

[7]Information about the latencies and throughputs can be found in manuals with instruction tables (Fog 2017; Intel 2018). APM883208-X1 (AppliedMicro X-Gene) implements ARM v8. Consequently, the latency of FMA for Cortex A57 can be used (ARM 2016). However, the throughput is different (Dolbeau 2016).

[8]The values in brackets were empirically determined and they are larger than the real latency values. This due to the trade-off between obtaining results that are optimal according to the architecture model (Section 4.5) and using as many of vector registers as possible. This problem can be solved by improving the register allocation (Section 4.2).

[9]The CPU frequencies with enabled Intel Turbo Boost technology are shown in brackets.

[10]If the -march=native option is not supported by the target architecture, then the -mcpu and -mtune options with appropriate values are used instead.

[11]In the cases of IBM Power 8 and DEGMM, the -disable-ppc-preinc-prep option in the LLVM's PowerPC backend is used.

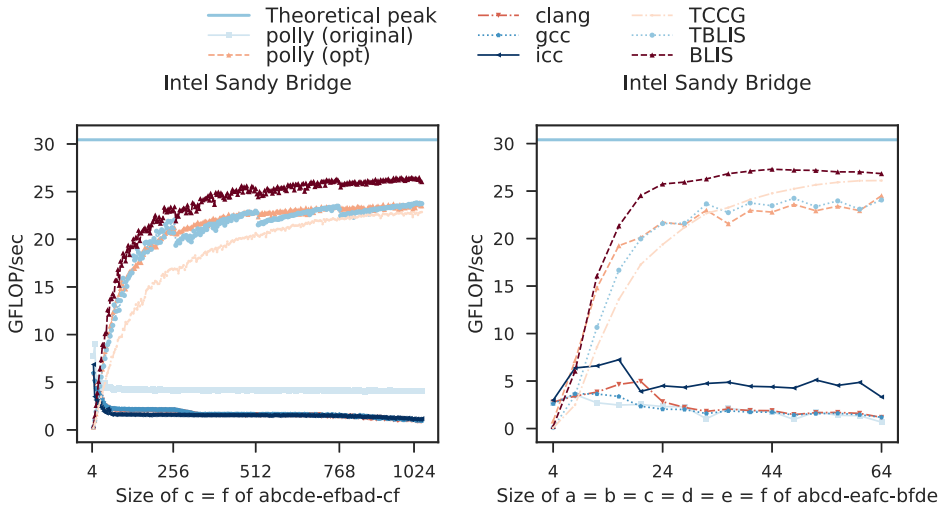

Fig. 4. TC for type double.

5.1 TC

We considered problems from a TC benchmark containing a wide range of user cases collected from previous studies related to TC (Springer and Bientinesi 2018), which included tensors with different dimensionality encountered in coupled-cluster methods (Stock et al. 2012) and quantum chemistry calculations (Baumgartner et al. 2005). We evaluated the single-threaded performance on the Intel Sandy Bridge machine for type double. To simplify the presentation, we encode a contraction $C_{\pi_C(IJ)} = \sum_P A_{\pi_A(IP)} \cdot B_{\pi_B(PJ)}$ as $\pi_C(IJ)$-$\pi_A(IP)$-$\pi_B(PJ)$. For instance, $C_{ijkl} = \sum_{m=0}^{n_m-1} \sum_{n=0}^{n_n-1} A_{imjn} \cdot B_{nlmk}$ can be encoded as ijkl-imjn-nlmk.

Figure 4 shows the evaluation results for abcde-efbad-cf and abcd-eafc-bfde, which are representative examples of TSc. The results for the full benchmark are provided in Appendix A. For abcde-efbad-cf, the sizes of the a, b, d, and e indices were fixed as 8, 8, 4, and 4, respectively. The sizes of the c and f indices were equal and they varied simultaneously, ranging from 4 to 1,024. In the case of abcd-eafc-bfde, the sizes of all the indices were equal and they varied simultaneously, ranging from 4 to 64. In addition, Figure 4 presents the evaluation results for MMA[×, +] with equivalent size and implemented in BLIS.

We observed substantial speedups when using the proposed approach compared with the production compilers, where it exceeded the reported ICC performance by up to 84× and outperformed GCC and Clang by up to 82×. However, we noted that if the tensors contained less than 32^2 elements, it was reasonable to use the default optimization techniques instead of the proposed approach and other approaches based on the mapping of TC to MMA[×, +] (e.g., TCCG (Springer and Bientinesi 2018) and TBLIS (Matthews 2016)). It corresponds to the results obtained for MMA[×, +] (Heinecke et al. 2015).

We conclude that the approach achieved more than 86.12% of the performance with TCCG[12] and TBLIS. In the next section, it is showed that this difference was caused by the suboptimal optimization of MMA[×, +], although the same performance was achieved in some cases. In the case of TCCG, the difference was due to the inherent disadvantages of TTGT (Hirata 2003) and LoG (Napoli et al. 2014) approaches, which are used to generate implementations of TCs, and,

[12]At the time of writing of this article, TCCG lacks an implementation of GETT for the Intel Sandy Bridge machine. Consequently, we only evaluated its implementations of LoG and TTGT.

subsequently, time them on the target platform. In particular, TTGT requires explicit tensor transpositions, which accounts for the pure overheads. LoG performs tensor slicing into a sequence of matrices, which can be small and/or incur strided memory accesses, thereby leading to poor performance. According to the results with BLIS, we note that BLIS exceeded the performance reported for TCCG by up to 1.5×.

In the case of TBLIS, the heuristics used to maximize the spatial locality of the data and reduce the number of cache and TLB misses can lead to suboptimal results. In particular, TBLIS logically reorders the dimensions from their original order so the dimensions in bundles I and J are sorted by increasing stride in the tensor that is produced by TC, e.g., C. Subsequently, the operand with the unit stride dimension of C is used as the first operand, e.g., A, for the mapped matrix-matrix multiplication and it is packed more frequently. However, it is not always possible to achieve unit stride access in both C and A using these heuristics (e.g., in the case of ijk-jli-lk with column-major order). The heuristics are modified in our approach. The LLVM's vectorization passes only store the second operand, e.g., B, of the mapped matrix-matrix multiplication to a vector register and load its elements after elements of A, so we swap A with B. The modified heuristics can also lead to suboptimal results. However, they allow us to keep the elements of B, which are reused more often than the elements of A, in the L1 cache. Furthermore, they can help to achieve the performance of TBLIS if its heuristics do not work (e.g., in the case of abcde-efbad-cf with row-major order).

5.2 MMA[×, +]

We consider the MMA[×, +] of the following form $C \leftarrow \alpha \otimes C \oplus \beta \otimes A \otimes B$,[13] implemented in the Polybench 3.2 benchmark suite (Pouchet 2011), which is a collection of programs from various domains that expose only the kernels that need to be optimized, and thus it is easy to work with.

Figures 5 and 6 present the results of single-threaded performance evaluations of the implementations of MMA[×, +] for type double and square and nonsquare matrices, respectively. Figure 7 shows the results for type float. We conclude that the optimization achieved a speedup of up to 20× compared with the modern production compilers and 1.63× compared with the Intel C compiler. We also note that if the matrices contained less than 80^2 elements, the optimization attained the performance of the single-threaded instances of MMA[×, +] available in MKL. However, if the matrices contained less than 32^2 elements, it was reasonable to use the default optimization techniques instead of the presented approach. It corresponds to Heinecke et al. (2015).

We found that in the case of Kaby Lake, square matrices, and type double, our approach achieved 83.33% of the performance of OpenBLAS. However, in the case of IBM Power 8, it only achieved 75.54%. These differences are explained by suboptimal register allocation, as mentioned previously (Section 4.2). Thus, we estimated the number of vector registers used in the case of Kaby Lake. According to Section 4.2, at each iteration of the innermost loop, $M_r N_r$ intermediate results are computed and $M_r + N_r$ elements of the operands of MMA[×, +] are used. Consequently, since only the elements of the second operand are stored in the vector registers (Section 4.2), at most $\frac{M_r N_r + N_r}{N_{VEC}}$ vector registers are used (Section 4.5). In the case of Kaby Lake, this number is equal to $\frac{8 \cdot 6 + 8}{4} = 12$, which is 87.5% of the vector registers are available on the machine. In the case of IBM Power 8, only 48.43% of the available vector registers are used. In the cases of Sandy Bridge and ARM, 62.5% and 65.62% of the available vector registers are used, respectively. Thus, Sandy Bridge

[13] Our implementation of the Algorithm 1 is based on LLVM's vectorization passes (Section 4), which do not vectorize multiplication of the parameters β and $A \otimes B$ at the time of writing this article. Furthermore, $\alpha \otimes C$ can be computed in a different SCoP statement, so our optimization does not influence its performance. Thus, different values for the parameters α and β only influence the vectorization but its optimization is not the goal of this study. For simplicity, we assume that their values are equal to one.

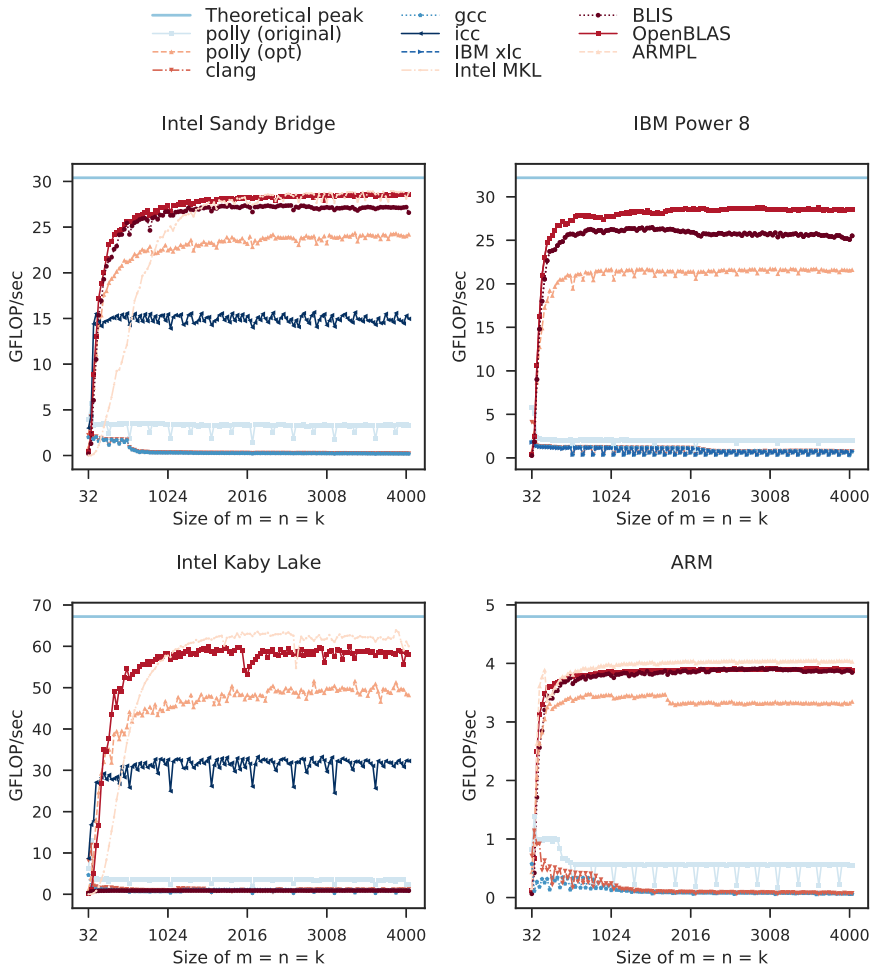

Fig. 5. MMA[×, +] of matrices that contain elements of type double.

and ARM could achieve 80.35% and 84.61% of the performance of OpenBLAS, respectively. We conclude that in the case of float type, Kaby Lake, IBM Power 8, Sandy Bridge, and ARM with our optimization method achieved 83.33%, 54.54%, 70%, and 86.25% of the performance of OpenBLAS, respectively, and this is due to the different numbers of vector registers used, i.e., 87.5%, 42.18%, 56.25%, and 65.62% in the cases of Kaby Lake, IBM Power 8, Sandy Bridge, and ARM, respectively.

We also evaluated the multi-threaded performance of the proposed approach for square matrices with dimensions of 8000 × 8000, type double, different values for the maximum number of threads in the OpenMP parallel region, and two target platforms. The first target platform evaluated had two sockets for 10-core Intel Xeon E5 CPUs (Table 1). The second had one 64-core Intel Xeon Phi CPU booted in flat mode (Table 1).

Figure 8 shows the results of the multi-threaded performance evaluations. In the case of Intel Xeon E5-2630, the proposed approach achieved 86.18% of the performance of the code from the optimized libraries. In the case of Intel Xeon Phi, the proposed approach exceeded the performance of OpenBLAS, which does not support Intel Xeon Phi CPUs and it uses a reference implementation. However, in the latter case, the proposed approach achieved only 69.02% of the performance of

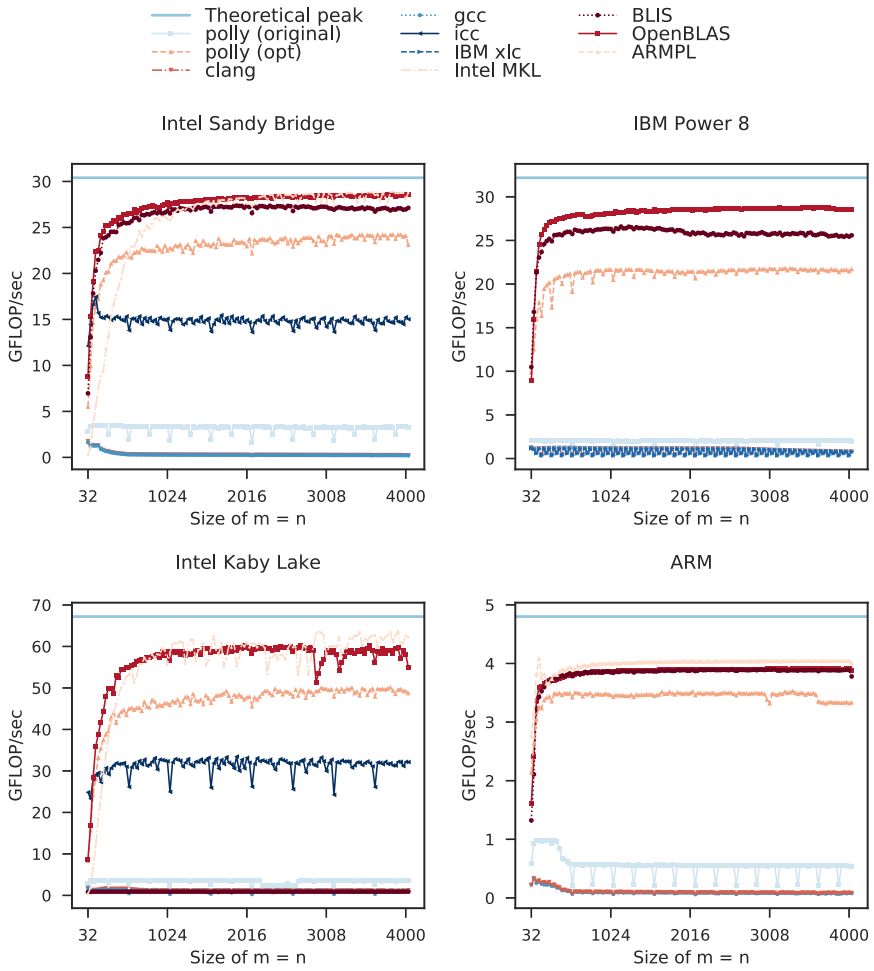

Fig. 6. MMA[×, +] of matrices that contain elements of type double (k = 2000).

BLIS due to the inability of Polly to parallelize i, which is the first loop around the macro-kernel. The Intel Xeon Phi cores are grouped in tiles, which comprise two cores with their own L2 cache. Consequently, i allows better parallelism than j_c, which is the second loop around the micro-kernel, and it could be parallelized by Polly (Section 4.6). We only considered the case of data level parallelism and a single-threaded implementation of a TC-like kernel, and further evaluations were beyond the scope of this study.

5.3 APPs

Since the 1970s, it has been known that MMA in different semirings can be used by solvers for a large number of problems combined under a single umbrella, which are called Algebraic Path Problem (APPs) (Sedukhin and Paprzycki 2012). We consider the following APPs: finding the least and the most reliable paths, finding paths with the maximum capacity, finding paths with the maximum cost, and finding for all pairs the shortest connecting path implemented using MMA[×, *min*], MMA[×, *max*], MMA[*min*, *max*], MMA[+, *max*], and MMA[+, *min*], respectively. We evaluated them for different types of elements (double and float) and square matrices.

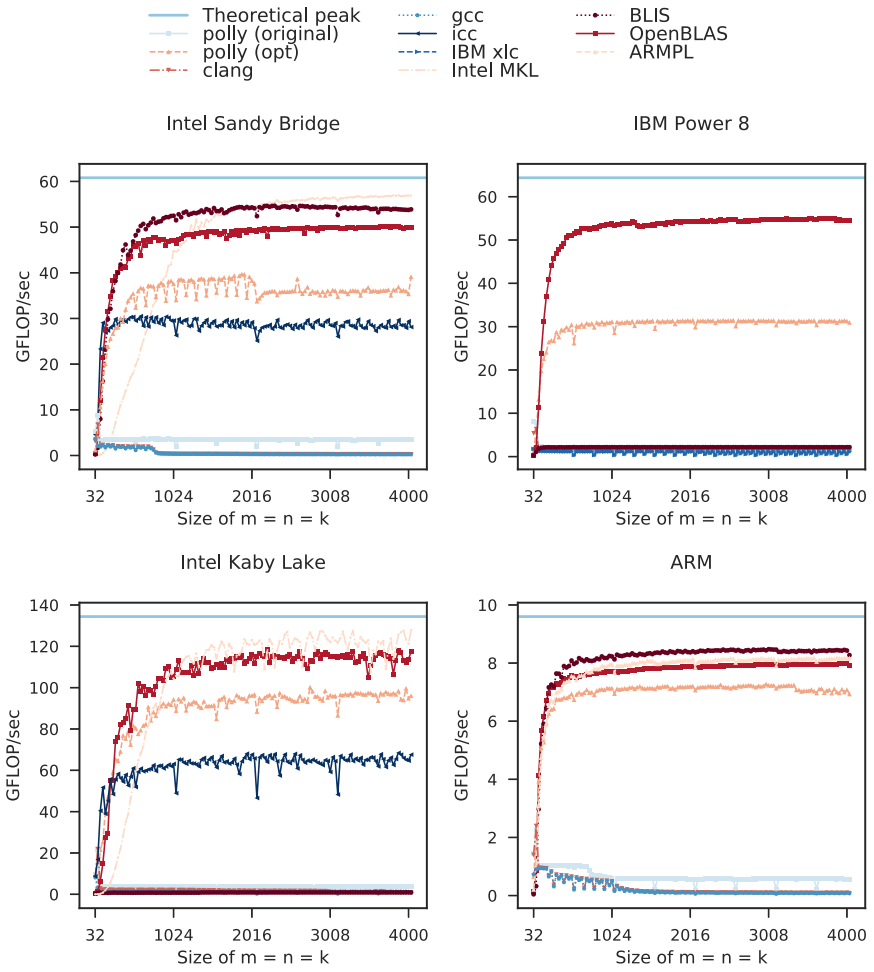

Fig. 7. MMA[×, +] of matrices that contain elements of type float.

The single-threaded performance evaluation results obtained for APPs on the Intel Sandy Bridge machine are shown in Appendix A. To compute the theoretical peak performance in GFLOP/sec for a particular MMA, we used the following standard formula (Ngxande and Moorosi 2014): theoretical peak performance = (CPU speed in GHz) × (CPU instruction per cycle). Consequently, the theoretical peak performance differed according to the ability of the CPU to simultaneously execute the micro-operations used to compute the MMA. For example, in the case of the double floating point data type, MMA[×, *min*], MMA[×, *max*], and MMA[×, −], 8 floating point operations could be executed per cycle. However, only 4 floating point operations could be executed per cycle in the case of MMA[+, *max*], MMA[+, *min*], and MMA[*min*, *max*]. This difference can be explained by the inability of Intel Sandy Bridge to simultaneously execute instructions that require the same execution unit (e.g., vmaxpd and vaddpd). In the case of MMA[/, *max*], only 0.4 floating point operations could be executed per cycle due to N_{VMMA}, which was equal to 0.05 (Section 4.5).

For the problems that required finding the least and the most reliable paths, the optimization achieved more than 69% of the theoretical peak performance and a speedup of 1.56× compared with the Intel C loop optimizer. For the other problems, the optimization achieved more than

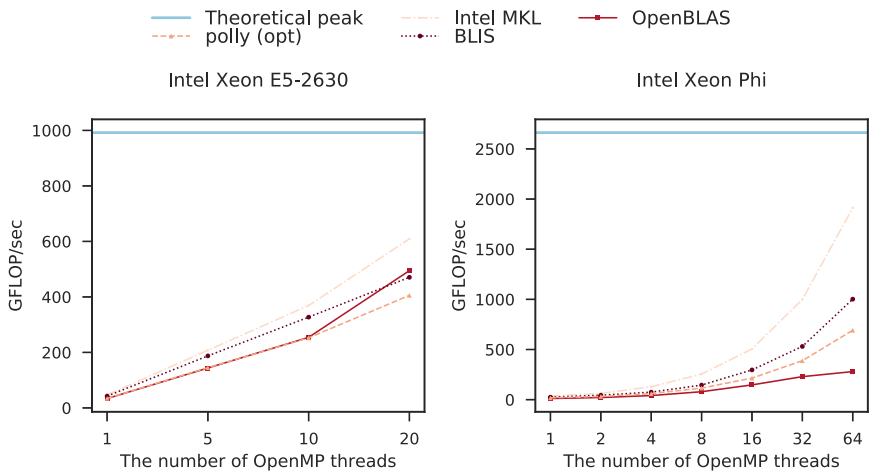

Fig. 8. MMA[×, +] for type double.

85% of the theoretical peak performance and produced similar results to the Intel C compiler. The performance of the code generated using our approach could be improved by applying prefetching instructions and optimized instruction scheduling (Section 4.2).

In addition to the performance evaluation results for APPs of type float, Appendix A shows the performance evaluation results for MMA[×, −] and MMA[/, *max*], which do not exist, since the corresponding binary operations do not form semirings. Nevertheless, the results show that we achieved 78% and 89% of the theoretical peaks in the cases of MMA[×, −] and MMA[/, *max*], respectively. Thus, the proposed approach can be applied even if the corresponding MMA operations do not form semirings. We also note that ICC allowed us to achieve high performance only for MMAs with throughputs less than one (e.g., MMA[+, *max*], MMA[*min*, *max*], MMA[+, *min*], and MMA[/, *max*] with Sandy Bridge).

6 RELATED WORK

In this section, we describe previous studies related to optimization of TC and the contribution of our proposed approach.

6.1 Automatic Optimization of MMA[×, +]

Approaches are available for fully automatically optimizing BLAS functions, particularly MMA[×, +]. In general, they are based on autotuning or domain-specific knowledge of algorithms for dense linear algebra kernels.

The Portable High-Performance ANSI C (PHiPAC) (Bilmes et al. 2014) and Automatically Tuned Linear Algebra Software (ATLAS) (Whaley et al. 2001) projects introduced autotuning for empirically determining the optimal parameters for BLAS routines when running them on the target platform. Autotuning can be combined with analytical techniques to identify the best optimization decisions, and this approach is used by LGen (Spampinato and Püschel 2014) and Build to Order BLAS (BTO) (Belter et al. 2009). LGen is a compiler for small-scale, basic linear algebra computations where the input is a fixed-size linear algebra expression and the output is a corresponding C function. BTO is a compiler where the input is a sequence of matrix and vector arithmetic statements in annotated MATLAB, and the output is a tuned implementation in C++. Autotuning can be time consuming and it can be difficult to apply it to production compilers where the execution time is a crucial factor.

Several approaches can fully automatically optimize MMA[×, +] and obtain high-performance code that outperforms expert-tuned implementations. The POET optimization library (Yi et al. 2014) and AUGEM framework (Wang et al. 2013) use annotations and templates of sequential code, respectively, written by domain experts to guide general-purpose compilers to produce optimized MMA[×, +] kernels from specifically prepared code. The Portable Compiler Approach (POCA) (Su et al. 2017) generates an optimized micro-kernel based on LLVM IR representing MMA[×, +] and subsequent domain-specific but architecture-independent optimizations of its micro-kernel. LIBXSMM (Heinecke et al. 2015) is a library that employs just-in-time in-memory code generation to obtain a high-performance implementation of small dense and sparse MMA[×, +] when high-performance libraries such as the Intel's MKL do not deliver optimal performance. The approaches for automaticall optimizing MMA[×, +] complement our method, allowing the creation of the optimization that helps to obtain the code that outperforms expert-tuned implementations.

6.2 Optimization of TC

Two main types of approaches can be applied to optimize TC (Springer and Bientinesi 2018): methods based on mapping TC to MMA[×, +] and methods based on nested loops.

The approaches based on nested loops (Apra et al. 2014) improve the performance of the high-level specifications of TCs by applying a series of loop transformations. However, their implementations can be degraded by strided memory access. The performance of small TCs that fit into caches can be improved using vectorization (Stock et al. 2011). In the case of larger TCs, loop-based code generators for GPUs can be used (Ma et al. 2011).

Transpose-Transpose-GEMM-Transpose (TTGT) (Hirata 2003) and Loops-over-GEMMs (LoG) (Napoli et al. 2014) use highly efficient MMA[×, +] implementations provided by BLAS-compatible libraries. In TTGT, explicit tensor transpositions "flatten" the arbitrary dimensional tensors into matrices to recast TC as a single MMA[×, +]. However, additional time is required to transpose the tensors as well as extra memory to store the transposition results. In LoG, tensors are sliced into a sequence of matrices that are contracted via MMA[×, +]. If the sliced matrices are small and/or the memory accesses produced are strided, then this can lead to poor performance. GEMM-like Tensor-Tensor multiplication (GETT) (Springer and Bientinesi 2018) and Tensor-Based Library Instantiation Software (TBLIS) (Matthews 2016) pack elements from the input tensors into fixed-size buffers to improve cache reuse and they subsequently use a micro-kernel that is typically implemented in assembly. Other than this similarity, there are several critical differences between GETT and BLIS. GETT uses an auto-fine-tuning framework guided by a performance model to implement TC for a fixed size. By contrast, TBLIS uses an entirely run-time algorithm, which can operate on any size of shape of tensor.

7 CONCLUSION AND FUTURE DIRECTIONS

In this study, we proposed a new compiler optimization to obtain highly optimized MMA and TC code without external software. Our approach is based on a new algorithm for the automatic transformation of TC, which is implemented[14] using a novel general infrastructure for performing arbitrary affine-linear data-layout transformations on the low-level compiler IR as well as the use of an analytical model.

We compared the execution time of code produced by optimizing MMA[×, +] with instances of MMA[×, +] that are available in Intel's MKL (Intel [n.d.]), ARMPL (ARM 2015), BLIS (Van Zee and van de Geijn 2015), and OpenBLAS (Xianyi et al. 2012), as well as code produced using the

[14]The extension of our optimization of MMA to TC is publicly available at https://gareevroman@bitbucket.org/gareevroman/polly-groman-fork.git.

current leading production compilers (GCC (Stallman 1999), ICC (Intel 2015), and IBM XL (IBM 2012)) and the compiler front-end Clang (Lattner 2002). Our method attained the performance of vendor optimized BLAS libraries, with a speedup of more than 1.63× compared with the state-of-art compilers. In the case of APPs, we achieved more than 85% of the theoretical peak performance and a speedup of 1.56× compared with the code produced by compilation. In the case of TC, we attained the performance of TCCG and TBLIS, and achieved 80.35% of the theoretical peak performance with a speedup of 84× compared with the Intel C loop optimizer.

We consider that our approach can be improved. In future research, we will map the proposed techniques to other BLAS operations. According to the expert multi-threaded implementation of MMA[×, +] (Smith et al. 2014), the code produced by the proposed optimization is suitable for parallelization. The algorithm could be improved by modifying the automatic vectorization. Optimizing sequences of BLAS functions obtains speedups of up to 137% compared with vendor optimized BLAS libraries (Belter et al. 2009) and this approach could also improve the proposed algorithm. Finding blocks that contain MMA may help to automatically transform algorithms such as Floyd–Warshall into a blocked form (Matsumoto and Sedukhin 2009; Takahashi and Sedukhin 2005), before their subsequent optimization.

APPENDIX

A FULL BENCHMARK

Table 3. Full Benchmark Results Using Double Precision[15]

Problem	Sizes	gcc (%)	clang (%)	icc (%)	polly (%)	polly (opt) (%)	Peak
MMA[+, max]	128	10.99	10.86	**87.83**	22.17	75.26	15.2
	1,024	2.17	2.17	**89.41**	12.3	88.42	
	2,000	1.84	1.64	89.87	21.91	**90.99**	
	4,000	1.32	1.32	89.67	21.97	**91.64**	
MMA[/, max]	128	42.11	42.76	81.58	42.76	**82.24**	1.52
	1,024	21.71	21.05	88.16	44.74	**88.16**	
	2,000	16.45	16.45	88.82	44.74	**88.82**	
	4,000	13.16	12.5	89.47	45.39	**89.47**	
MMA[min, max]	128	10.07	11.05	**94.14**	21.91	76.84	15.2
	1,024	2.11	2.11	88.03	11.71	**88.16**	
	2,000	1.64	1.64	88.16	21.84	**92.04**	
	4,000	1.32	1.25	90.92	21.51	**92.43**	
MMA[×, max]	128	5.26	5.39	47.76	11.02	**56.18**	30.4
	1,024	1.09	1.05	46.94	6.15	**76.61**	
	2,000	0.82	0.86	47.63	10.86	**76.97**	
	4,000	0.66	0.66	48.42	10.92	**79.14**	
MMA[×, min]	128	5.33	5.43	49.54	11.15	**55.69**	30.4
	1,024	1.05	1.05	47.53	5.92	**76.94**	
	2,000	0.82	0.82	46.94	10.85	**77.43**	
	4,000	0.66	0.63	48.59	10.76	**78.52**	

(Continued)

Table 3. Continued

Problem	Sizes	gcc (%)	clang (%)	icc (%)	polly (%)	polly (opt) (%)	Peak
MMA[×, −]	128	5.33	5.43	48.36	11.18	**55.03**	30.4
	1,024	1.09	1.05	46.64	6.09	**76.38**	
	2,000	0.82	0.82	47.8	10.89	**77.73**	
	4,000	0.63	0.66	47.34	11.05	**78.52**	
MMA[+, min]	128	10.99	10.99	**88.36**	22.57	76.78	15.2
	1,024	2.11	2.11	**89.21**	12.04	88.42	
	2,000	1.64	1.64	91.18	21.91	**91.38**	
	4,000	1.25	1.25	**90.92**	21.78	90.79	

Table 4. Full Benchmark Results Using Single Precision[15]

Problem	Sizes	gcc (%)	clang (%)	icc (%)	polly (%)	polly (opt) (%)	Peak
MMA[+, max]	128	6.05	6.15	**91.94**	11.38	70.99	30.4
	1,024	1.09	1.09	84.01	6.18	**89.28**	
	2,000	1.02	1.02	87.73	11.58	**89.38**	
	4,000	0.79	0.79	86.61	11.55	**89.34**	
MMA[/, max]	128	23.68	23.68	80.92	23.68	**90.13**	3.04
	1,024	10.86	11.18	89.47	24.01	**94.41**	
	2,000	10.2	10.2	88.49	24.01	**95.07**	
	4,000	7.89	7.89	90.79	24.01	**95.39**	
MMA[min, max]	128	5.33	6.18	**90.89**	11.35	71.22	30.4
	1,024	1.09	1.09	85.26	6.15	**89.47**	
	2,000	1.02	1.02	87.27	11.64	**89.74**	
	4,000	0.79	0.79	86.41	11.58	**90.39**	
MMA[×, max]	128	2.93	2.93	47.71	5.66	**62.38**	60.8
	1,024	0.54	0.54	43.19	3.11	**62.38**	
	2,000	0.51	0.51	46.3	5.74	**61.38**	
	4,000	0.39	0.39	46.05	5.76	**62.23**	
MMA[×, min]	128	2.93	2.98	**47.6**	5.67	43.37	60.8
	1,024	0.54	0.54	43.32	3.06	**56.74**	
	2,000	0.51	0.51	46.88	5.71	**62.1**	
	4,000	0.39	0.39	46.73	5.76	**62.86**	
MMA[×, −]	128	2.91	2.98	**46**	5.74	42.35	60.8
	1,024	0.54	0.54	43.63	3.11	**58.38**	
	2,000	0.51	0.51	47.27	5.74	**62.83**	
	4,000	0.39	0.39	46.25	5.77	**59.77**	
MMA[+, min]	128	6.09	6.18	**91.94**	11.38	70.99	30.4
	1,024	1.08	1.08	84.01	6.18	**89.28**	
	2,000	1.02	1.02	87.73	11.58	**89.38**	
	4,000	0.79	0.79	86.61	11.55	**89.34**	

Table 5. Full Benchmark Results Using Double Precision[15]

TC[16]	Sizes	GCC (%)	Clang (%)	ICC (%)	Polly (original) (%)	Polly (opt) (%)	TBLIS (%)	TCCG (%)
ab-ac-cb	1,024×1,024-1,024×1,024-1,024×1,024	1.05	1.05	44.84	5.69	74.21	77.27	**85.82**
ab-acd-dbc	1,024×1,024-1,024×32×32-32×1,024×32	1.12	1.12	2.5	1.09	72.57	77.2	**81.78**
abc-acd-db	32×1,024×32-32×32×1,024-1,024×1,024	1.05	1.05	7.4	12.07	77.2	78.52	**82.96**
abc-ad-bdc	1,024×32×32-1,024×1,024-32×1,024×32	5	4.87	13.55	26.61	74.8	77.93	**82.37**
abc-adc-bd	32×1,024×32-32×1,024×32-1,024×1,024	5.46	5.53	8.36	32.93	77.76	**79.18**	78.62
ab-cad-dcb	1,024×1,024-32×1,024×32-32×32×1,024	1.02	1.02	17.07	0.99	75.07	77.14	**81.25**
abc-adec-ebd	32×1,024×32-32×32×32×32-32×1,024×32	1.05	1.05	17.76	1.05	**78.75**	78.07	74.54
abc-adc-db	32×1,024×32-32×1,024×32-1,024×1,024	0.92	0.92	25.1	30.79	76.48	**78.62**	78.49
abc-bda-dc	32×32×1,024-32×1,024×32-1,024×1,024	0.92	0.92	32.07	13.91	74.77	76.22	**82.53**
abcd-aebf-dfce	32×32×32×32-32×32×32×32-32×32×32×32	1.05	0.95	1.09	5.07	72.07	76.15	**78.71**
abcd-aebf-fdec	32×32×32×32-32×32×32×32-32×32×32×32	1.02	0.95	0.99	0.99	72.5	72.34	**79.01**
abcd-aecf-bfde	32×32×32×32-32×32×32×32-32×32×32×32	5.59	6.61	6.78	5.89	76.64	**78.06**	75.39
abcd-aecf-fbed	32×32×32×32-32×32×32×32-32×32×32×32	1.09	4.11	14.74	1.05	74.54	**78.29**	74.93
abcd-aedf-bfce	32×32×32×32-32×32×32×32-32×32×32×32	5.76	16.94	13.45	6.02	77.73	**77.83**	75.56
abcd-aedf-fbec	32×32×32×32-32×32×32×32-32×32×32×32	1.09	11.81	15.79	1.05	76.78	**78.12**	75.36
abcd-aefb-fdce	32×32×32×32-32×32×32×32-32×32×32×32	0.99	0.95	5	1.02	70.56	72.39	**78.22**
abcd-aefc-fbed	32×32×32×32-32×32×32×32-32×32×32×32	5.03	16.97	26.05	2.76	78.22	**78.39**	75.1
abc-dca-bd	32×1,024×32-1,024×32×32-1,024×1,024	1.02	1.02	15.66	6.25	76.41	78.06	**82.01**
abcd-dbea-ec	16×8×1,024×8-8×8×1,024×16-1,024×1,024	1.74	1.78	3.13	9.11	**77.89**	77.4	77.76
abcd-deca-be	16×1,024×8×8-8×1,024×8×16-1,024×1,024	1.94	1.97	6.74	5.36	77.17	77.11	**78.95**
abcd-ea-ebcd	1,024×16×8×8-1,024×1,024-1,024×16×8×8	3.65	3.68	10.92	18.29	74.11	77.17	**86.05**
abcd-eafb-fdec	32×32×32×32-32×32×32×32-32×32×32×32	1.02	0.95	5.26	0.89	72.04	71.98	**77.3**
abcd-eafc-bfde	32×32×32×32-32×32×32×32-32×32×32×32	5.23	5.95	14.74	3.42	67.5	**77.57**	73.88
abcd-eafd-fbec	32×32×32×32-32×32×32×32-32×32×32×32	1.09	12.89	26.61	0.92	**78.22**	78.19	72.96
abcd-ebad-ce	16×8×1,024×8-1,024×8×16×8-1,024×1,024	5.16	5.13	1.71	22.86	80.33	79.21	**82.17**
abcd-eb-aecd	16×1,024×8×8-1,024×1,024-16×1,024×8×8	4.08	4.21	2.7	26.25	76.88	**78.68**	78.36
abcd-ec-abed	16×8×1,024×8-1,024×1,024-16×8×1,024×8	5	5	1.25	30.59	**79.47**	79.08	77.01
abcde-ecbfa-fd	8×8×4×1,024×4-4×4×8×1,024×8-1,024×1,024	9.24	14.7	2.8	14.41	**78.52**	73.26	75.63
abcde-efbad-cf	8×8×1,024×4×4-4×1,024×8×8×4-1,024×1,024	3.29	3.09	0.95	13.39	**80.3**	78.39	75.59
abcde-efcad-bf	8×1,024×8×4×4-4×1,024×8×8×4-1,024×1,024	3.29	3.16	3.45	13.29	**78.88**	78.28	76.91
abcdef-dega-gfbc	16×16×8×8×8×8-8×8×1,024×16-1,024×8×16×8	2.73	2.76	1.78	5.1	74.7	76.55	**76.78**
abcdef-degb-gfac	16×16×8×8×8×8-8×8×1,024×16-1,024×8×16×8	2.7	2.7	1.74	5.3	74.61	76.48	**77.57**
abcdef-degc-gfab	16×16×8×8×8×8-8×8×1,024×8-1,024×8×16×16	4.84	4.84	2.66	3.95	70.76	72.27	**78.98**
abcdef-dfga-gebc	16×16×8×8×8×8-8×8×1,024×16-1,024×8×16×8	3.06	3.03	3.62	7.14	71.94	76.25	**76.55**
abcdef-dfgb-geac	16×16×8×8×8×8-8×8×1,024×16-1,024×8×16×8	2.99	2.93	4.05	7.07	72.83	76.28	**77.99**
abcdef-dfgc-geab	16×16×8×8×8×8-8×8×1,024×8-1,024×8×16×16	3.78	3.88	3.72	6.32	72.47	71.94	**77.57**
abcdef-efga-gdbc	16×16×8×8×8×8-8×8×1,024×16-1,024×8×16×8	1.02	1.02	3.52	4.47	72.07	76.25	**76.88**
abcdef-efgb-gdac	16×16×8×8×8×8-8×8×1,024×16-1,024×8×16×16	1.02	1.02	10.49	4.47	72.5	**76.44**	76.38
abcdef-efgc-gdab	16×16×8×8×8×8-8×8×1,024×8-1,024×8×16×16	1.02	1.02	3.65	4.74	71.55	71.38	**77.14**
abcdef-gdab-efgc	16×16×8×8×8×8-1,024×8×16×16-8×8×1,024×8	0.99	0.99	3.62	4.38	71.45	71.61	**76.84**
abcdef-gdac-efgb	16×16×8×8×8×8-1,024×8×16×8-8×8×1,024×16	0.99	0.99	10.49	3.49	72.43	76.02	**76.58**

(Continued)

Table 5. Continued

TC[16]	Sizes	GCC (%)	Clang (%)	ICC (%)	Polly (original) (%)	Polly (opt) (%)	TBLIS (%)	TCCG (%)
abcdef-gdbc-efga	16×16×8×8×8×8-1024×8×16×8-8×8×1024×16	1.02	1.02	3.49	4.7	70.33	76.25	**77.11**
abcdef-geab-dfgc	16×16×8×8×8×8-1024×8×16×16-8×8×1024×8	3.78	3.75	3.72	6.12	72.37	71.61	**78.32**
abcdef-geac-dfgb	16×16×8×8×8×8-1024×8×16×8-8×8×1024×16	3.03	3.16	4.05	6.88	73.06	75.33	**78.26**
abcdef-gebc-dfga	16×16×8×8×8×8-1024×8×16×8-8×8×1024×16	2.99	2.99	3.62	6.81	71.97	76.15	**76.97**
abcdef-gfab-degc	16×16×8×8×8×8-1024×8×16×16-8×8×1024×8	4.24	4.47	2.7	3.88	70.79	72.6	**78.95**
abcdef-gfac-degb	16×16×8×8×8×8-1024×8×16×8-8×8×1024×16	2.66	2.76	1.74	4.87	74.9	77.04	**77.17**
abcdef-gfbc-dega	16×16×8×8×8×8-1024×8×16×8-8×8×1024×16	2.63	2.86	1.78	4.8	74.18	**76.68**	76.18

ACKNOWLEDGMENTS

We thank all our anonymous reviewers for critically reading the manuscript and suggesting substantial improvements such as optimization of TC. We also thank the Swiss National Supercomputing Centre (CSCS) for providing HPC resources and ARM's Manchester Design Center for fruitful discussions about micro-kernels.

REFERENCES

Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo et al. 2015. TensorFlow: Large-scale machine learning on heterogeneous systems. Retrieved from https://www.tensorflow.org/.

E. Apra, M. Klemm, and K. Kowalski. 2014. Efficient implementation of many-body quantum chemical methods on the Intel E Xeon Phi coprocessor. In *Proceedings of the International Conference for High-Performance Computing, Networking, Storage, and Analysis (SC'14)*. 674–684. DOI: http://dx.doi.org/10.1109/SC.2014.60

ARM. 2015. *ARM Performance Libraries Reference Manual*. ARM.

ARM. 2016. *Cortex-A57 Software Optimization Guide*. ARM.

Rodney J. Bartlett and Monika Musiał. 2007. Coupled-cluster theory in quantum chemistry. *Rev. Mod. Phys.* 79 (2007), 291–352. Issue 1. DOI: http://dx.doi.org/10.1103/RevModPhys.79.291

G. Baumgartner, A. Auer, and D. Bernholdt. 2005. Synthesis of high-performance parallel programs for a class of *ab initio* quantum chemistry models. *Proc. IEEE* 93, 2 (Feb. 2005), 276–292. DOI: http://dx.doi.org/10.1109/JPROC.2004.840311

Geoffrey Belter, E. R. Jessup, Ian Karlin, and Jeremy G. Siek. 2009. Automating the generation of composed linear algebra kernels. In *Proceedings of the Conference on High-Performance Computing Networking, Storage and Analysis (SC'09)*. ACM, New York, NY. DOI: http://dx.doi.org/10.1145/1654059.1654119

Jeff Bilmes, Krste Asanovic, Chee-Whye Chin, and Jim Demmel. 2014. Optimizing matrix multiply using PHiPAC: A portable, high-performance, ANSI C coding methodology. In *Proceedings of the ACM International Conference on Supercomputing 25th Anniversary*. ACM, 253–260.

Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. 2008. A practical automatic polyhedral parallelizer and locality optimizer. *SIGPLAN Not.* 43, 6 (June 2008), 101–113. DOI: http://dx.doi.org/10.1145/1379022.1375595

Jason Cong and Bingjun Xiao. 2014. *Minimizing Computation in Convolutional Neural Networks*. Springer, Cham, 281–290. DOI: http://dx.doi.org/10.1007/978-3-319-11179-7_36

Romain Dolbeau. 2016. Theoretical peak FLOPS per instruction set on less conventional hardware. https://www.researchgate.net/publication/308804090_Theoretical_Peak_FLOPS_per_instruction_set_on_less_conventional_hardwar. (Accessed: July 10, 2018).

J. J. Dongarra, Jeremy Du Croz, Sven Hammarling, and I. S. Duff. 1990. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Softw.* 16, 1 (Mar. 1990), 1–17. DOI: http://dx.doi.org/10.1145/77626.79170

F. Facchinei, S. Sagratella, and G. Scutari. 2014. Flexible parallel algorithms for big data optimization. In *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP'14)*. 7208–7212. DOI: http://dx.doi.org/10.1109/ICASSP.2014.6854999

[15]Numbers represent percentage of performance relative to theoretical peak performance. Theoretical peak performance is specified in GFLOP/sec.

[16]The presented tensors are stored in row-major order.

Paul Feautrier and Christian Lengauer. 2011. *Polyhedron Model.* Springer US, Boston, MA, 1581–1592. DOI: http://dx.doi.org/10.1007/978-0-387-09766-4_502

Agner Fog. 2017. *Instruction Tables.* Technical University of Denmark.

Sylvain Girbal, Nicolas Vasilache, Cédric Bastoul, Albert Cohen, David Parello, Marc Sigler, and Olivier Temam. 2006. Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies. *Int. J. Parallel Program.* 34, 3 (June 2006), 261–317. DOI: http://dx.doi.org/10.1007/s10766-006-0012-3

Kazushige Goto and Robert A. van de Geijn. 2008. Anatomy of high-performance matrix multiplication. *ACM Trans. Math. Softw.* 34, 3, Article 12 (May 2008). DOI: http://dx.doi.org/10.1145/1356052.1356053

Tobias Grosser, Armin Größlinger, and Christian Lengauer. 2012. Polly—Performing polyhedral optimizations on a low-level intermediate representation. *Parallel Process. Lett.* 22, 4 (2012).

Tobias Grosser, Hongbin Zheng, Ragesh Aloor, Andreas Simbürger, Armin Größlinger, and Louis-Noël Pouchet. 2011. Polly—Polyhedral optimization in LLVM. In *Proceedings of the 1st International Workshop on Polyhedral Compilation Techniques (IMPACT'11)*, C. Alias and C. Bastoul (Eds.). Chamonix, France.

Robert Harrison, Gregory Beylkin, Florian Bischoff et al. 2016. MADNESS: A multiresolution, adaptive numerical environment for scientific simulation. *SIAM J. Sci. Comput.* 38, 5 (2016), S123–S142.

Alexander Heinecke, Hans Pabst, and Greg Henry. 2015. LIBXSMM: A high-performance library for small matrix multiplications. In *Proceedings of the International Conference for High-Performance Computing, Networking, Storage and Analysis (SC'15).*

Tom Henretty, Kevin Stock, Louis-Noël Pouchet, Franz Franchetti, J. Ramanujam, and P. Sadayappan. 2011. Data layout transformation for stencil computations on short-vector SIMD architectures. In *Proceedings of the 20th International Conference on Compiler Construction: Part of the Joint European Conferences on Theory and Practice of Software (CC'11/ETAPS'11).* Springer-Verlag, Berlin, 225–245.

So Hirata. 2003. Tensor contraction engine: Abstraction and automated parallel implementation of configuration-interaction, coupled-cluster, and many-body perturbation theories. *J. Phys. Chem. A* 107, 46 (2003), 9887–9897. DOI: http://dx.doi.org/10.1021/jp034596z

IBM 2012. *XL C/C++: Compiler Reference—IBM.* IBM.

Intel. [n.d.]. Intel Math Kernel Library (Intel MKL). Retrieved from https://software.intel.com/ru-ru/intel-mkl/?cid=sem43700011401059448&intel_term=intel+mkl&gclid=CIjbtvqaqM8CFSoNcwodDPUAbw&gclsrc=aw.ds.

Intel. 2015. *Intel C++ Compiler 16.0 Update 4 User and Reference Guide.* Intel.

Intel. 2018. *Intel Intrinsics Guide.* Intel.

Shana Jayachandran and T. Venkatachalam. 2016. A secure scheme for privacy preserving data mining using matrix encoding. *World Eng. Appl. Sci. J.* 7, 3 (2016), 190–193. DOI: http://dx.doi.org/10.5829/idosi.weasj.2016.7.3.22525

Chris Lattner. 2002. *LLVM: An Infrastructure for Multi-Stage Optimization.* Master's thesis. Computer Science Department, University of Illinois at Urbana-Champaign, Urbana, IL. Retrieved from http://llvm.cs.uiuc.edu.

C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. 1979. Basic linear algebra subprograms for fortran usage. *ACM Trans. Math. Softw.* 5, 3 (Sept. 1979), 308–323. DOI: http://dx.doi.org/10.1145/355841.355847

Michael Lehn. 2014. GEMM: From Pure C to SSE Optimized Micro Kernels. Retrieved from http://apfel.mathematik.uni-ulm.de/~lehn/sghpc/gemm/index.html.

Vincent Loechner and Doran K. Wilde. 1997. Parameterized polyhedra and their vertices. *Int. J. Parallel Program.* 25, 6 (1997), 525–549. DOI: http://dx.doi.org/10.1023/A:1025117523902

Tze Meng Low, Francisco D. Igual, Tyler M. Smith, and Enrique S. Quintana-Orti. 2016. Analytical modeling is enough for high-performance BLIS. *ACM Trans. Math. Softw.* 43, 2, Article 12 (Aug. 2016). DOI: http://dx.doi.org/10.1145/2925987

Wenjing Ma, Sriram Krishnamoorthy, Oreste Villa, and Karol Kowalski. 2011. GPU-based implementations of the noniterative regularized-CCSD(T) corrections: Applications to strongly correlated systems. *J. Chem. Theory Comput.* 7, 5 (2011), 1316–1327. DOI: http://dx.doi.org/10.1021/ct1007247

Kazuya Matsumoto and Stanislav G. Sedukhin. 2009. A solution of the all-pairs shortest paths problem on the cell broadband engine processor. *IEICE Trans.* 92-D, 6 (2009), 1225–1231.

Devin Matthews. 2016. High-performance tensor contraction without BLAS. *CoRR* abs/1607.00291.

Vijay Menon and Keshav Pingali. 1999. High-level semantic optimization of numerical codes. In *Proceedings of the 13th International Conference on Supercomputing (ICS'99).* ACM, New York, NY, 434–443. DOI: http://dx.doi.org/10.1145/305138.305230

Edoardo Di Napoli, Diego Fabregat-Traver, Gregorio Quintana-Ortí, and Paolo Bientinesi. 2014. Towards an efficient use of the BLAS library for multilinear tensor contractions. *Appl. Math. Comput.* 235 (2014), 454–468. DOI: http://dx.doi.org/10.1016/j.amc.2014.02.051

Mkhuseli Ngxande and Nyalleng Moorosi. 2014. Development of Beowulf cluster to perform large datasets simulations in educational institutions. In *International Journal of Computer Applications* 99, 15 (Aug. 2014), 29–35.

Dmitry Pekurovsky. 2012. P3DFFT: A framework for parallel computations of fourier transforms in three dimensions. *SIAM J. Sci. Comput.* 34, 4 (2012), C192–C209. DOI: http://dx.doi.org/10.1137/11082748X

Louis-Noël Pouchet. 2011. PolyBench/C the Polyhedral Benchmark suite. Retrieved from http://web.cse.ohio-state.edu/~pouchet/software/polybench/.

William Pugh and David Wonnacott. 1994a. *An Exact Method for Analysis of Value-Based Array Data Dependences*. Springer, Berlin, 546–566. DOI: http://dx.doi.org/10.1007/3-540-57659-2_31

William Pugh and David Wonnacott. 1994b. Static analysis of upper and lower bounds on dependences and parallelism. *ACM Trans. Program. Lang. Syst.* 16, 4 (July 1994), 1248–1278. DOI: http://dx.doi.org/10.1145/183432.183525

Stanislav G. Sedukhin and Marcin Paprzycki. 2012. Generalizing matrix multiplication for efficient computations on modern computers. In *Proceedings of the 9th International Conference on Parallel Processing and Applied Mathematics (PPAM'11)*. Springer-Verlag, Berlin, Heidelberg, 225–234. DOI: http://dx.doi.org/10.1007/978-3-642-31464-3_23

Tyler M. Smith, Robert van de Geijn, Mikhail Smelyanskiy, Jeff R. Hammond, and Field G. Van Zee. 2014. Anatomy of high-performance many-threaded matrix multiplication. In *Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium (IPDPS'14)*. IEEE Computer Society, Washington, DC, 1049–1059. DOI: http://dx.doi.org/10.1109/IPDPS.2014.110

Daniele G. Spampinato and Markus Püschel. 2014. A basic linear algebra compiler. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO'14)*. 23–32.

Paul Springer and Paolo Bientinesi. 2018. Design of a high-performance GEMM-like tensor-tensor multiplication. *ACM Trans. Math. Softw.* 44, 3, Article 28 (Jan. 2018). DOI: http://dx.doi.org/10.1145/3157733

Y. N. Srikant and P. Shankar. 2007. *The Compiler Design Handbook: Optimizations and Machine Code Generation, Second Edition*. CRC Press.

R. Stallman. 1999. *Using and Porting the GNU Compiler Collection: For Gcc-2.95*. Free Software Foundation.

Kevin Stock, Tom Henretty, Iyyappa Murugandi, P. Sadayappan, and Robert Harrison. 2011. Model-driven SIMD code generation for a multi-resolution tensor kernel. In *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium (IPDPS'11)*. IEEE Computer Society, Washington, DC, 1058–1067. DOI: http://dx.doi.org/10.1109/IPDPS.2011.101

Kevin Stock, Louis-Noël Pouchet, and P. Sadayappan. 2012. Using machine learning to improve automatic vectorization. *ACM Trans. Archit. Code Optim.* 8, 4, Article 50 (Jan. 2012). DOI: http://dx.doi.org/10.1145/2086696.2086729

Xing Su, Xiangke Liao, and Jingling Xue. 2017. Automatic generation of fast BLAS3-GEMM: A portable compiler approach. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO'17)*. IEEE Press, Piscataway, NJ, 122–133.

Akihito Takahashi and Stanislav Sedukhin. 2005. *Parallel Blocked Algorithm for Solving the Algebraic Path Problem on a Matrix Processor*. Springer, Berlin, 786–795. DOI: http://dx.doi.org/10.1007/11557654_89

H. M. Tufo and P. F. Fischer. 1999. Terascale spectral element algorithms and implementations. In *Proceedings of the ACM/IEEE Conference on Supercomputing (SC'99)*. ACM, New York, NY. DOI: http://dx.doi.org/10.1145/331532.331599

Field G. Van Zee and Robert A. van de Geijn. 2015. BLIS: A framework for rapidly instantiating BLAS functionality. *ACM Trans. Math. Softw.* 41, 3, Article 14 (June 2015). DOI: http://dx.doi.org/10.1145/2764454

Nicolas Vasilache, Jeff Johnson, Michaël Mathieu, Soumith Chintala, Serkan Piantino, and Yann LeCun. 2014. Fast convolutional nets with fbfft: A GPU performance evaluation. *CoRR* (2014). Retrieved from http://arxiv.org/abs/1412.7580.

Qian Wang, Xianyi Zhang, Yunquan Zhang, and Qing Yi. 2013. AUGEM: Automatically generate high-performance dense linear algebra kernels on x86 CPUs. In *Proceedings of the International Conference on High-Performance Computing, Networking, Storage and Analysis (SC'13)*. ACM, New York, NY. DOI: http://dx.doi.org/10.1145/2503210.2503219

M. Watson, R. Olivares-Amaya, R. G. Edgar, and A. Aspuru-Guzik. 2010. Accelerating correlated quantum chemistry calculations using graphical processing units. *Comput. Sci. Eng.* 12, 4 (July 2010), 40–51. DOI: http://dx.doi.org/10.1109/MCSE.2010.29

R. Clint Whaley, Antoine Petitet, and Jack J. Dongarra. 2001. Automated empirical optimizations of software and the ATLAS project. *Parallel Comput.* 27, 1 (2001), 3–35. DOI: http://dx.doi.org/10.1016/S0167-8191(00)00087-9

Zhang Xianyi, Wang Qian, and Zhang Yunquan. 2012. Model-driven level 3 BLAS performance optimization on Loongson 3A processor. In *Proceedings of the IEEE 18th International Conference on Parallel and Distributed Systems (ICPADS'12)*. IEEE, 684–691.

Qing Yi, Qian Wang, and Huimin Cui. 2014. Specializing compiler optimizations through programmable composition for dense matrix computations. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'14)*. IEEE Computer Society, Washington, DC, 596–608. DOI: http://dx.doi.org/10.1109/MICRO.2014.14

Received October 2017; revised May 2018; accepted June 2018